Gemmell Górriz, Carlos (2026) *Externalising reasoning by teaching language models to use programs as tools.* PhD thesis.

https://theses.gla.ac.uk/86006/

Externalising Reasoning by Teaching Language Models to Use Programs as Tools

Carlos Gemmell Górriz

Submitted in fulfilment of the requirements for the
Degree of Doctor of Philosophy

School of Engineering
College of Science and Engineering
University of Glasgow

February 2026

Abstract

Large Language Models (LLMs) have revolutionised Natural Language Processing (NLP) by scalably identifying patterns in text to predict subsequent tokens with minimal human supervision. However, unlike humans who readily employ external tools such as calculators and spreadsheets to manage and reason over large-scale structured information, LLMs are typically self-contained, relying solely on their learned weights to reason over the input context. This limitation hinders their performance on tasks that require precise, scalable reasoning over structured data, where intuitive pattern matching often fails. This thesis argues that LLMs can overcome this limitation by learning to write and utilise computer programs as tools, mirroring human strategies of externalising reasoning to enhance accuracy on tasks involving structured information.

To substantiate this claim, this thesis first establishes that transformer-based LLMs can translate natural language (NL) into short programs. It further demonstrates that retrieval mechanisms can boost this accuracy by incorporating relevant code snippets. Recognising the challenge of acquiring extensive training data for robust program generation, this work investigates the use of pre-trained LLMs as a source of weak supervision. This research demonstrates that aligning pre-training and fine-tuning objectives through a process termed "reiteration" significantly benefits NL-to-code translation. Furthermore, accuracy is enhanced by leveraging the unsupervised generative capabilities of LLMs to create synthetic aligned data pairs of natural language and code.

Subsequently, this thesis demonstrates the concept of tool use by leveraging the code-generation capabilities of language models in the context of semi-structured data. It empirically establishes that LLMs struggle with question answering over increasingly large tables. To address this deficiency, this thesis proposes a novel framework wherein LLMs generate task-specific, executable programs to filter rows. The output from these programs is then fed back into the LLM, exemplifying the use of the tool. I show that the application of these program tools significantly improves the accuracy of LLMs when querying large, structured datasets by enabling them to scalably remove irrelevant information. Finally, the thesis presents a practical, real-world application of tool use, showcasing the effectiveness of language-conditioned code writing in a conversational assistant setting. In this setting, the assistant interprets a user's utterance, generates corresponding code, executes it to update its internal state, and generates a response.

At each turn in the conversation, the system reintroduces the output from the previously executed code to the code-writing LLM. This continuous tool use allows for fluid conversation over long, structured tasks, thereby validating the broader applicability of the tool-augmented LLM paradigm.

Contents

List of Tables

List of Figures

Acknowledgements

Had it not been for several key people, this thesis would have not been possible. I'm grateful to my PhD colleagues Federico Rossetto, Iain Mackie, Paul Owoicho, Sophie Fisher, and Alessandro Spegiorin. I would like to thank my supervisors Jeff Dalton and Sean MacAvaney, who helped me through thoughtful discussion to express my ideas. Lastly, I'm grateful to my dad who helped me see the vision clearer than I could have myself.

Declaration

All work in this thesis was carried out by the author unless otherwise explicitly stated.

Chapter 1

Introduction

This introductory chapter lays the foundation for the thesis by establishing the core technical problem, outlining the proposed solution, and providing a conceptual framework for understanding its contributions. First, it will discuss the inherent limitations of encoding knowledge purely within the parameters of a neural network. It will then argue that augmenting these models with the ability to generate and execute external programs offers a powerful path forward. Finally, it will detail how this approach enables more robust and complex reasoning.

1.1 A Conceptual Framework: Humans and Tools

Tools are so fundamental to the human species that the materials used to create them define historical periods — such as the Stone Age, Bronze Age, Iron Age, and the Information Age [55]. In essence, tools extend the brain and body beyond their biological limits.

The human brain is a remarkable feat of biological and cultural evolution; however, it has a limited capacity for storing and processing information. Humans can overcome these limitations by utilising tools. For instance, we use calculators for arithmetic, maps for navigation, calendars to remember appointments, and photographs to recall our personal past [13, 19, 35]. These tools possess informational properties that assist in performing cognitive tasks. The use of tools often enhances our cognitive capacities, making tasks easier, faster, more reliable, or even possible in the first place.

As noted by Lawick-Goodall [77], tool-using performance in animals is "the use of an external object as a functional extension [...] in the attainment of an immediate goal". Broadly speaking, a tool is a means to an end, and its use represents an intermediate step in what can be an elaborate process. Although other species use tools [26, 55], humans are the most extensive users. We possess dedicated neural systems for understanding the causal relationship between using a tool and achieving a goal, a faculty that likely underpins all technological development [102].

Despite their clear benefits, tools are not useful in isolation. Rather, it is the combination of a user and a tool that is effective. This highlights an important point: a tool exposes an interface for its use, and the user must learn to operate within the parameters of that interface. Humans are not born with an innate knowledge of how to use tools [55]. Indeed, mastering the relevant interfaces for any tool, from sharpening rocks to programming computers, can take years of practice [5,14,28]. When integrated successfully, however, the brain can represent a tool as part of the body itself, allowing the user to use it as if it were a physical extension of the body [90].

This effective combination also poses the question of how it can be modelled computationally. This thesis proposes a firm step towards this goal. However, to do so, we must first narrow the scope of tools under study. Given the wide variety of tools humans use, this thesis narrows its definition of a *tool* to focus specifically on informational tasks involving semi-structured data. I focus on semi-structured data because humans already exhibit clear strategies to circumvent their innate limitations with informational tools. A clear example of this type of tool use is how a person processes tabular data, a central task that will be explored in Chapter 6. For instance, consider a person gathering dietary requirements for a group of people to estimate food quantities:

- When faced with a **simple table**, such as for a house party, a person might process the information "in situ," keeping a mental tally. This process involves mentally filtering guests, counting variables, and aggregating the results to create a shopping list.
- When faced with a **large, complex table**, such as for a well-attended international conference, the same person would likely turn to a tool like a spreadsheet program to filter, sort, and aggregate the data with keywords, making the intermediate steps more manageable. However, successfully completing the task requires proficiency with the spreadsheet program to ensure the results are trustworthy.

This example illustrates a conceptual framework for tool-enabled systems: given an input data structure *(a list of attendees and their dietary requirements)* and a target task *(to estimate food quantities)*, the system decides whether to use a tool. If the system chooses a tool, it uses the appropriate interface *(a tabular program)* to interact with the data. The system then interprets the output of the tool *(a newly transformed spreadsheet)* in the context of the original task to produce the desired output *(a shopping list)*. This framework, which guides all contributions in this thesis, is outlined in further detail in Chapter 2.

Existing Natural Language Processing (NLP) systems are reasonably capable of solving small-scale tasks like the house party example [16,64]. I further demonstrate this in Chapter 6, where they struggle significantly to answer questions on tabular data as the scale and complexity of the data increase. To understand the reasons for this struggle and how to address it, we must first examine the limitations of current NLP systems.

1.2 The Limits of Parametric Knowledge

Artificial neural networks are parametric function approximators that map an input vector to an output vector through a composition of linear transformations and pointwise nonlinearities. A set of learnable parameters, or weights, entirely controls their behaviour. While early neural networks were inspired by biological neurons [92], the modern view is purely computational. Given a dataset of input-output pairs, the training process tweaks the parameters to minimise a differentiable loss function over the dataset via gradient-based optimisation [118].

Researchers extended this paradigm to sequential data with the advent of Recurrent Neural Networks (RNNs) and their more advanced variant, Long Short-Term Memory (LSTM) networks, which introduced gated state updates to learn longer-range dependencies [58]. Further improvements came with sequence-to-sequence models incorporating attention mechanisms, which enhanced generalisation in tasks like machine translation [4, 129]. A significant architectural shift occurred with the Transformer, which replaced recurrence with self-attention, allowing for parallel processing of all tokens in a sequence and enabling efficient training at an unprecedented scale [133]. This innovation led to the development of Large Language Models, trained on the simple objective of next-token prediction [32, 112]. During inference, an LLM processes a sequence of text divided into tokens and auto-regressively predicts the subsequent token to generate a text completion [123].

Language models following this pattern have achieved great success across numerous NLP tasks, including reading comprehension, question answering, translation, and summarisation [112]. A landmark example of this scaling potential is GPT-3, a 175-billion-parameter transformer that demonstrates sophisticated pattern-matching capabilities, allowing users to program it in English during inference using text instructions known as prompts [16]. This scaling paradigm has been so effective that increasing data and model size has become the primary lever for improving performance in NLP. Despite these successes, all such models possess inherent limitations that further scaling can only partially address.

A long-standing and well-documented issue with neural networks is their lack of compositional generalisation [38, 69, 75, 120]. This is often associated with a failure to extrapolate, meaning they struggle to perform effectively on sequences longer than those encountered during training. Research has shown that LLMs, in particular, perform poorly on this class of problems [9, 98, 138]. Furthermore, improvements on one compositional generalisation benchmark do not necessarily transfer to others, limiting their utility as general-purpose models for real-world tasks that cannot be solved with a small set of rules [40].

Another significant limitation of LLMs is the static nature of their knowledge. Since the neural network cannot update the weights during inference, LLMs are unable to access up-to-date information about recent events [74]. This static knowledge base also contributes to their tendency to hallucinate facts [63, 91]. The training process optimises the next-token prediction objective for generating plausible, fluent sequences, not necessarily factually correct or logically

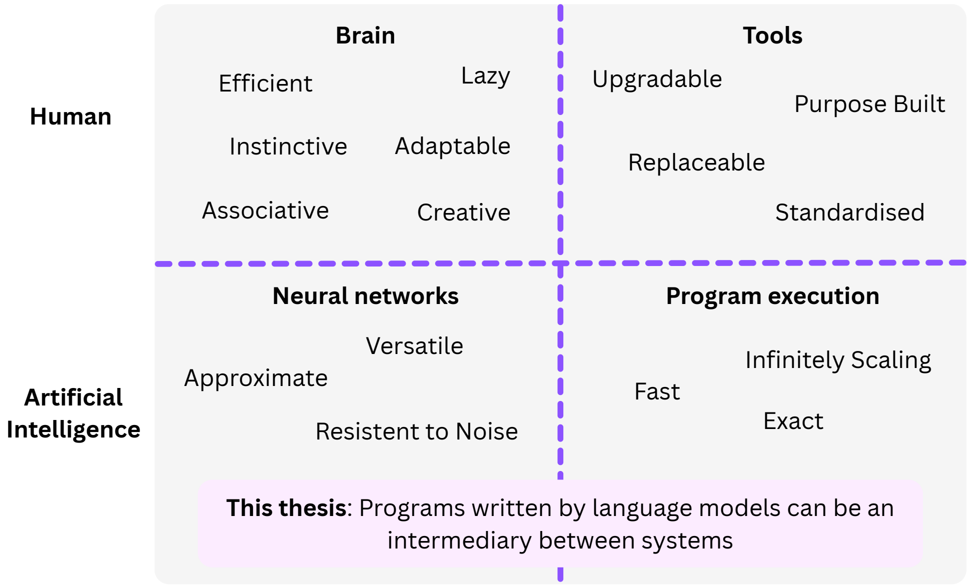


Figure 1.1: A visualisation of this thesis's contribution as a bridge within artificial intelligence. We conceptualise the strengths of language models as analogous to the human brain (pattern recognition, fluency), and the strengths of program execution as external tool use (infinitely scalable, precise calculation, logical execution). This thesis focuses on bridging these two modes for tasks involving semi-structured data.

sound ones. Consequently, hallucination and mathematical errors are natural outcomes of an objective that prioritises fluency over veracity.

The architecture of LLMs also imposes practical constraints. They have a limited context length, often a maximum of a few thousand tokens, which restricts the amount of input information the system can condition them on at one time [16]. Furthermore, LLMs lack the mathematical skills required for precise calculations [107]. Standard sequence-to-sequence models often fail to solve simple math word problems, and even when specifically trained for arithmetic, their ability to generalise does not extend beyond the number of digits seen during training [50]. This difficulty with precision extends to information retrieval at scale; for instance, performance on a simple key-value retrieval task degrades as one places the relevant information further back in the context window [87].

Despite these issues, LLMs can execute simple, language-framed instructions, demonstrating a basic reasoning capability [16]. This is analogous to a human who can perform simple mental arithmetic like 5+6 but requires a calculator for $\sqrt{59}$. I can extend this parallel: both humans and LLMs have limitations related to context length, reliance on heuristics, and a lack of precision at scale. Conversely, their strengths also align, including adaptability, resistance to noisy input, and intuitive pattern matching. In fact, we can see an LLM's limitations in precise reasoning as the inverse of its greatest strength: fuzzy pattern matching and tolerance for ambiguity. This quality is what makes LLMs adept at understanding messy human language and intent, where a traditional, rigid program would fail.

From this analogy, as shown in Figure 1.1, we can extrapolate that tools might serve a similar purpose for LLMs as they do for humans: augmenting their native capabilities in areas where they are not well-suited. For informational tasks involving semi-structured data, this thesis proposes that LLMs can act as users of tools, capable of both generating the correct commands to interact with them and understanding their output. The goal is to decouple the task of fuzzy, natural language generation from the task of precise, scalable reasoning. The central question then becomes how to achieve this synthesis.

1.3 Programs as the Solution: Bridging Large Language Models and Informational Tools

The idea of combining neural and symbolic approaches is not new, as seen in concepts like Neural Turing Machines [50]. However, the underlying neural architectures lacked the necessary expressive power for wider popularisation. In these models, the network calls symbolic operations directly from neural activations, making adaptation and generalisation difficult. The situation was akin to programming a modern video game on a 1980s computer; the core ideas were present, but the hardware and software abstractions were too primitive to be effective.

A critical advancement that helps bridge this gap is the shift towards a unified text-to-text representation for many NLP tasks [114]. The simple but powerful idea of treating instructions, raw data, code, and outputs as plain text creates a universal interface. This paradigm contrasts sharply with earlier methods that treated code as a distinct modality, such as Abstract Syntax Networks, which processed it as abstract nodes [111], or Neural Turing Machines, which did not process code at all. When we view all components as text, the tool use framework becomes significantly less coupled to specific tools, their arguments, or the mechanics of program generation. Instead, it places more emphasis on the tool user's ability to understand language syntax and write effective code.

Within this text-to-text framework, high-level programming languages like Python emerge as an ideal interface for scalable computations. Humans use these languages to express complex sequences of transformations on semi-structured data in a way that is both concise and expressive.

This thesis proposes that executable programs are the ideal external tool to augment LLMs for several key reasons:

- **They are deterministic and precise,** directly counteracting the probabilistic, approximate nature of language models.

- **They are inherently procedural and scalable,** perfectly suited for the algorithmic tasks that language models often fail at, such as precise filtering of large datasets or multi-step arithmetic.

- **They are composable and expressive,** allowing the system to construct and execute complex, multi-step reasoning reliably, far beyond what can be achieved through a single forward pass of a neural network.

Prior work in semantic parsing has demonstrated that neural networks can translate natural language requirements into a limited set of formal operators, paving the way for generating more complex outputs [7]. It is important, however, to draw a clear distinction between traditional semantic parsing and the concept of tool use as framed in this thesis. While semantic parsing focuses on the translation from language to a program, this thesis conceptualises tool use as a means to an end, implying a decision-making process about whether and how a tool should be employed.

I do not envision tool use as a simple "translate-and-execute" process. As detailed in Chapter 2, this thesis outlines a multi-step framework: 1) The system first understands the task in the context of the available semi-structured data. 2) It then decides whether using a tool is necessary. 3) If so, the system generates a program expressing the desired reasoning. 4) The system executes this program on the data, yielding an output. 5) Finally, the system formulates an answer to the original task based on this new, transformed data.

A key insight of this thesis is that modern language models are capable of generating these programs themselves and interpreting their output. The research presented in Chapters 4 and 5 shows that models can learn to translate a natural language description of intent into syntactically correct and semantically appropriate Python snippets, representing the first crucial step towards tool use. Subsequently, Chapters 6 and 7 fully instantiate this framework, demonstrating the scale and precision that the combination of language models and external program execution can achieve on the task of semi-structured tabular question answering.

1.4 Contemporary Relevance and Enduring Limitations

The work presented in this thesis was carried out between 2019 and 2023, when the idea of a language model generating and executing programs as tools was a research hypothesis rather than an engineering default. Since then, this architectural principle has moved from the fringes of NLP research to the foundation of the AI industry. The launch of GitHub Copilot in 2021 marked the first mass-market deployment of AI-generated code. Orchestration frameworks such as LangChain and the Model Context Protocol (MCP) followed, standardising how language models discover and invoke external tools [3]. Most compellingly, frontier reasoning models, OpenAI's o1 and o3, DeepSeek-R1, and Google's Gemini, now embed Python execution environments directly within their reasoning traces. These systems, built with far greater resources than those studied here, independently arrived at the central insight of this thesis: that externalising computation to programs allows language models to perform reasoning they cannot reliably achieve alone.

Some of the specific limitations documented here have been addressed by scale and improved training data. The models studied in Chapters 4 and 5 required carefully designed training procedures to generate syntactically correct code; this is less true today. Adding code to pre-training data measurably improves this as the gains between Llama 1 and Llama 2 when code was introduced to the training mixture show [130,131]. The results in this thesis should therefore be read as principled early evidence for an architectural approach whose value was not yet obvious, not as a precise characterisation of current limitations.

The fundamental premise of this thesis, however, has not been eliminated by scale: language models still struggle to reason over large, semi-structured data. Liu et al. showed that performance degrades systematically when relevant information appears in the middle of a long context, and this holds across modern frontier models [87]. In agentic settings, Kate et al. find 7–91% degradation in answer retrieval as tool response length increases [68], a direct parallel to the table-size degradation demonstrated in Chapter 6. The industry's growing reliance on embedded program execution is the clearest evidence of all: the problem has not been solved, only widely enough recognised to become the default approach to addressing it.

This thesis therefore retains its salience beyond its historical moment. The question it addresses, whether language models can generate and use programs to externalise reasoning they are fundamentally ill-suited for, remains as pressing in 2026 as it was when this work began and possibly for years to come. The thesis statement that follows should be understood in this light.

1.5 Thesis Statement

In this thesis, I postulate that language models are capable of writing programs and using them as tools with a similar effect to that of humans to externalise reasoning at scale to improve accuracy in semi-structured information tasks.

We can decompose this statement into two arguments:

Language Models can Generate Code

This thesis states that we can make natural language-conditioned program generation more accurate using transformer-based sequence-to-sequence models, and, in particular, transformers combined with a retrieval component. Moreover, general-purpose language models can act as sources of weak-supervision data that further enhance natural language to code performance, and specifically, that language infilling objectives are superior for this task compared to the traditional translation objectives.

Language Models can Use Executable Code as a Tool

I claim that the ability of transformer-based language models to generate executable programs from natural language can be used to improve their accuracy, specifically in tabular question answering. This is achieved by combining the scalable and exact execution of row filtering programs over the input table with the language model's neural table understanding capabilities.

1.6 Contributions

Having formally stated the thesis statement, this section breaks it down into concrete, testable hypotheses and details the specific contributions I make in each chapter to validate them.

The ability to semantically translate between the space where a user poses a task (English) and an environment where the system can leverage a tool (code) is a prerequisite for effective tool usage. As a result, we formulate Hypotheses 1 and 2 to prove the first part of the thesis statement.

Proving that language models can use code as a tool requires a task with structure and natural language. We use tabular question answering as the setting to formulate Hypothesis 3, where the large and semi-structured nature of tables proves an effective setting to demonstrate the effectiveness of programs to mediate between an approximate neural system and exact execution.

Hypothesis 1: *NL to code translation can be improved through a combination of the Transformer architecture and retrieval.*

The first contribution my thesis makes is in Chapter 4 in the domain of natural language (NL) to code translation. At the time of writing, researchers approach NL-to-code translation as a structured prediction task with domain-specific architectures. After presenting the task, I demonstrate the differences in modelling capabilities for several approaches: long short-term memory networks, transformers, and retrieval, and I show that the Transformer is capable of implicitly modelling code syntax better than other general-purpose methods. I also find that combining the Transformer with a fixed retrieval tool improves translation accuracy compared to traditional approaches in this task.

> **Hypothesis 1.1** *Transformer architectures are an effective method over LSTMs and retrieval for the task of NL to code translation.*
>
> In Section 4.4, I show that existing non-AST methods for code translation can be inadequate, particularly in retrieval, where even an oracle retriever yields sub-optimal results. This highlights the requirement for a more flexible form of generation. These systems are traditionally LSTM-based. After outlining the baseline results described in Section 4.2, I empirically demonstrate the transformer architecture's superior ability at modelling the task of natural language to code translation. This architecture outperforms all prior sequence-to-sequence-based methods across a variety of Python-based tasks.

Hypothesis 1.2 *Retrieval, as a fixed tool, improves the effectiveness of transformer architectures for NL to code translation.*

While transformer architectures themselves demonstrate superior ability as established under Hypothesis 1.1, their effectiveness can be further enhanced. Indeed, retrieval methods have long been shown to be effective in augmenting various NLP tasks by incorporating similar, relevant documents or snippets from an external corpus. As stated before, retrieved results on their own are not effective in this task. However, in Section 4.2.4 I demonstrate that we can effectively combine retrieved results with a transformer to improve overall translation accuracy over any standalone method. Even with a fixed tool like a retrieval system, this evidence shows that language models can benefit from externally provided information from executed programs. Importantly, not all methods of including retrieved results improve performance equally. In Section 4.4, empirically, I show how my proposed method of biasing decoding externally is superior to naïvely concatenating search results with the input in this task.

Hypothesis 2: *NL to code translation can be improved through sources of weak supervision.*

Chapter 5 of my thesis examines the benefits of weak supervision signals derived from pre-trained language models in a natural language to code translation setting. Specifically in the context of BART [81], a pre-trained language model, I examine the role of two distinct methods: continued pre-training with infilling objectives to approximate the original causal language modelling objective, and synthetic data derived from pre-trained language models.

Hypothesis 2.1 *Infilling training objectives are superior to translation objectives for NL to code translation.*

To begin exploring how weak supervision can improve NL-to-code translation (Hypothesis 2), a primary consideration is the choice of training objective when adapting pre-trained models. Traditionally, one adapts a pre-trained language model to a new task by modelling the new objective directly. First, I prove the effectiveness of infilling fine-tuning on a single modality in the context of conversational query rewriting in Section 5.2. Then I transfer the method, in Section 5.3.3, for natural language to code translation. I find that infilling objectives can be more effective than direct translation. Specifically, I show that using continued infilling during pre-training and fine-tuning to leverage unsupervised signals can lead to more accurate code translation.

Hypothesis 2.2 *Infilling pre-training that jointly considers aligned natural language (NL) and code modalities is more effective than independent modality pre-training for NL to code generation.*

Armed with the knowledge that infilling pre-training is an effective method to condition a BART language model for natural language to code translation, I study the impact data modality mixes have on performance in Section 5.4. Specifically, I examine the effect our proposed pre-training has when I consider natural language and code modalities independently and jointly in weak supervision scenarios. I empirically show that jointly pre-training on aligned weak supervision sources of code and natural language is most effective for translation similarity and parsing accuracy, followed in performance by solely pre-training on code.

Hypothesis 2.3 *Language models can be used to generate weak supervision data to improve the effectiveness of NL to code translation systems.*

In the final sections of Chapter 5, I examine the use of pre-trained language models as generators of synthetic data directly for the task of natural language to code translation. Specifically, I study their effectiveness in generating the natural language description of an unlabelled code snippet. In Section 5.5, I show that pre-trained language models can be effective in improving knowledge and syntax for downstream models by reporting improved translation performance when using the synthetic data than without.

Hypothesis 3: *Language models can generate question-specific programs to improve question answering accuracy from tabular data.*

After confirming the ability of language models to translate code snippets from natural language in Chapters 4 & 5, I close the loop in Chapter 6 and use these abilities for tabular question answering (TQA). TQA provides an ideal setting to evaluate the abilities of language models to reason over semi-structured data scalably. Specifically, I examine the use of programs to filter tabular data, enabling language models to process it more effectively, which leads to increased question-answering accuracy.

Hypothesis 3.1 *Performance of language-model-based TQA systems is influenced by table size and can be improved by selectively filtering tabular data inputs.*

To substantiate the claim that language models can effectively use generated code as tools for tasks like tabular question answering (Hypothesis 3), we must first establish the baseline challenges these models face. Therefore, in Section 6.3, I demonstrate that, for a variety of language-model-based TQA systems, question answering accuracy negatively correlates with input table size. I show that the trend exists before the system truncates information due to transformer context window sizes, indicating a fundamental limitation for direct language-model-based tabular question answering. I further confirm this in Section 6.3.2 by manually filtering table rows, reducing their size, and showing an increase in performance irrespective of question difficulty.

Hypothesis 3.2 *In TQA settings, when filter operations are applicable, language models can generate query-specific programs that, when executed, improve the accuracy of neural TQA systems.*

Armed with the knowledge that language-model-based TQA systems benefit from table row filtering, in Section 6.4.1, I investigate the effectiveness of language models to produce programs that, once executed, can filter table rows agnostic to the table size. I study various methods to generate programs based on different pre-trained language models. The empirical findings in Section 6.5.1 show that, under specific conditions, programmatic filters produced by language models improve downstream model accuracy.

Hypothesis 3.3 *Language model signals can be used to identify when to apply query-specific programs automatically, thus leading to improved accuracy in neural TQA systems.*

As I previously established, row filtering operations are beneficial for many questions in TQA settings, though not for all. In Section 6.5.2 I study the effectiveness of various query specific signals to identify when applying a row filtering program will lead to accuracy benefits. Following empirical analysis in Section 6.5.3, I show that language models display useful signals as a consequence of processing challenging queries, specifically, the sequence log probability of the produced answer.

These hypotheses combined allow us to validate the thesis statement that language models are capable of writing programs and using them as tools to externalise reasoning at scale, thereby improving accuracy in semi-structured information tasks.

1.7 Origins of Materials

Having detailed the specific contributions and hypotheses of this thesis, it is necessary to acknowledge the provenance of these ideas and findings. Most of the material presented in this thesis is based on work considered by various international conferences and workshops:

- **Chapter 4**: Relevance Transformer and the study into the use of transformers and retrieval for natural language to code generation is accepted at **SIGIR 2020** [47].
- **Chapter 5**: I present the work at the start of Chapter 5 on infilling pre-training for conversational rewriting as a notebook paper at **TREC 2020** [43]. The studies on methods for pre-training natural language to code translation are not, as of yet, reviewed.
- **Chapter 6**: ToolWriter and my investigations into the use of programs as tools to improve the accuracy of language models in tabular question answering are accepted at **EMNLP 2023** [44].
- **Chapter 7**: Shows the use of programs generated by language models in a practical use case in GRILLBot, the winning system for the 2021/22 Amazon Alexa Prize TaskBot competition. The work is accepted at **SIGDial 2022** [46] and presented as a technical Amazon workshop paper [45]. The co-authors of the GRILLBot team approved of the work attributed to me from the aforementioned cited papers for this thesis. Specifically, all work on the NDP, the intent classification system, the QA system and the system architecture can be entirely attributed to me.

1.8 Organisation

To guide the reader through the subsequent chapters, this section outlines the overall organisation of the thesis. It is important to note that, given the fast pace of progress in the area of NLP and language modelling, I present the results in each section as they were in the context at the time of original publication. This means that I do not add new baselines retroactively to studies. Many of these works predate the arrival of large language models and were at the forefront of performance. I organise the remainder of this thesis as follows:

- Chapter 2 presents a framework for tool use that helps put the contributions of this thesis in context. In the first part, I describe the high-level component groups that help build intuition for how we can model tool use computationally. Then I outline its sufficiency. Finally, I step through each component in the framework, describing its function.

- Chapter 3 describes the background of sequence-to-sequence models, semantic parsing, core retrieval techniques, and approaches to integrate programs and neural networks. First, I describe the isolated learning process that leads neural networks for text to uncover patterns in data. Following this, I cover the use of language modelling as a powerful unsupervised method that leads to *emergent capabilities* in sequence-to-sequence models. I then describe how semantic parsing uses programs as a means to derive an answer from structured data through execution. Afterwards, I show early approaches to the integration of both neural and programmatic approaches to simple problems. Finally, I describe approaches to conversational assistance to put tool use in Chapter 7 in context.

- Chapter 4 presents support for Hypothesis 1. I present the task of translation from natural language to code with a specific focus on sequence-to-sequence methods and retrieval. Through evaluation on three collections of parallel English and Python code, I uncover that transformers are an effective method for the task over prior baselines. I explore the role retrieval can play in this task and contribute a new method for this task that combines both transformers and pseudo-relevance feedback to improve the performance of base transformers for this task.

- Chapter 5 describes the importance of pre-training and the use of weak supervision signals to improve effectiveness in the task of natural language to code translation. Specifically, I study infilling objectives and demonstrate that, in this task, it is superior to the commonly used translation objective. I provide evidence for this in two tasks: code translation and conversational query rewriting. I show that this infilling objective is well-suited to leverage weak supervision signals from parallel corpora and synthetic labels from large language models. I empirically demonstrate on the CoNaLa dataset that a combination of these methods can surpass domain-specific methods for code generation.

- Chapter 6 presents a method to unify language models and programs for the task of tabular question answering (TQA). I first present a study on the performance of language-model-based systems on long tables, finding sharp decreases in performance as table size increases. This leads us to propose programs as a means to reduce table size. I show that automatically generated query-specific programs are effective in several TQA benchmarks. Our proposed approach is language model agnostic and provides further evidence that programs acting as tools can enhance language models.

- Chapter 7 provides an applied case of automatically generated programs being used as a means of translating user intent into a complex action-space. I show this in the context of GRILLBot, the prize-winning conversational agent developed for the Amazon Alexa Prize TaskBot competition. Specifically, I present the neural decision parser as a sequence-to-sequence model that jointly processes system state with user utterances to produce a system policy expressed as Python code. This stands as a practical use case where programs act as a compositional medium to express complex intent.

- Chapter 8 closes this thesis by highlighting the contributions and the conclusions of each chapter. I also discuss future directions for research.

Chapter 2

Tool use framework

This chapter formalises the "Human-Tool" analogy from the introduction into a concrete computational framework. It serves as a unifying lens through which the contributions can be understood. It deconstructs the process of augmenting a language model with external tools into distinct, interacting components. First, this chapter provides an overview of the functionally distinct groups within the framework. Following this, a step-by-step description details how these components interact to solve a task and contextualises the specific contributions of this thesis.

2.1 High-Level Groups

The framework comprises four conceptual groups to solve an informational task on semi-structured data:

- **The Task:** In this thesis, we focus on text-based informational tasks with associated semi-structured data. The task definition also includes metrics to measure the performance of the system against a ground truth target.
- **The Data:** This refers to a predefined, semi-structured data source that contains the information necessary to complete the task. This could be a table of conference attendees or the internal state of a conversational agent. This data structure is the object upon which the tool operates and is formatted to be passed to later neural components.
- **The Agent:** At the risk of anthropomorphising, this is the collection of systems that make decisions on how to interpret the data and take necessary action in the context of the task to solve it. This can include calling tools and interpreting the output. In this thesis, we separate the tasks that the overall tool-using agent needs to do into separate components to make improvements on each in isolation. For instance, by creating these boundaries between components, we can understand a necessary nuance between tool use and code generation, with the latter being a central point of study in Chapters 4 & 5.

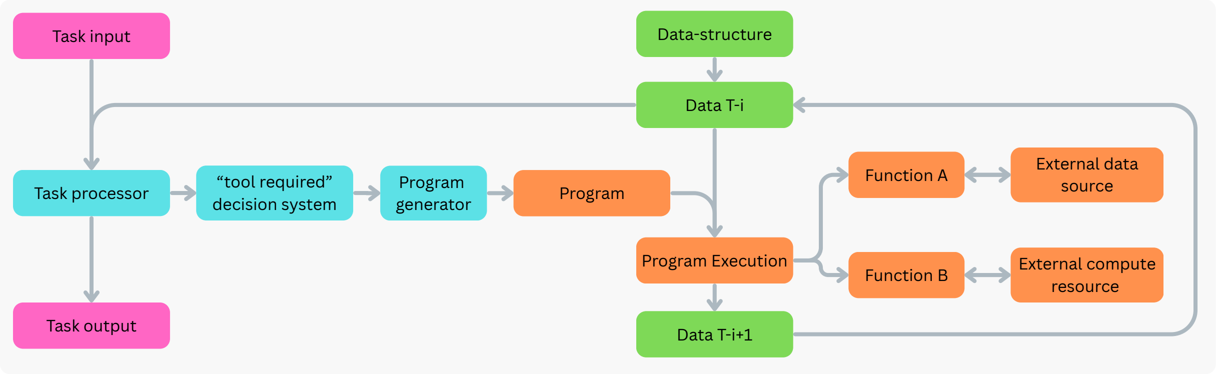


Figure 2.1: The tool use framework used throughout this thesis, with all components colour-coded by conceptual group. Components for **the task (in purple)**, **the data (in green)**, **the agent (in blue)**, and **the tool (in orange)**.

- **The Tool:** This is a system, separate from the agent, that exposes an interface for the agent to use. The tool is a means to an end and exhibits characteristics that are complementary to the agent, such that by using the tool, the agent may be able to augment its capabilities. In this thesis, the tool is the collection of logical operators that form an executable environment that is programmable through a text-based programming language. It is important to note that "the program" isn’t the tool but actually the interface to transform data via the digital executable environment. This digital, discrete, and scalable quality of executed programs is ultimately the desirable property that complements neural network-based language models.

Before detailing each component, it is important to clarify the scope and sufficiency of the proposed framework. The framework is designed to capture informational tasks involving semi-structured data—data that combines rigid and free-form dimensions. Typical examples include tables, key–value stores, and JSON-like schemas that preserve consistent fields while allowing flexible content. Although this framework does not aim to represent all forms of tool use exhaustively, it is sufficient for modelling tasks that require access, transformation, or reasoning over semi-structured information, such as tabular question answering or state manipulation in conversational systems.

Its sufficiency derives from two key properties. First, the framework makes no assumptions about the limitations of the task processor, meaning it can, in principle, solve any informational task over semi-structured data. However, as later chapters demonstrate, practical systems like large language models reveal significant limitations when operating without other framework elements—particularly when they attempt to reason directly over unfiltered or complex data. Second, the framework is sufficiently expressive for programmatic tool use because it makes no assumptions about the complexity of the generated program. However, the framework constrains the effects of program execution to operations within the given data structure for the purpose of solving the task.

2.2 Individual Component Descriptions

Armed with the knowledge of how the core abstractions interact, the following sections step through the components of the framework.

Task Input: This is the informational task the system must complete in the context of the data. For instance, in the introductory example of estimating food quantities given dietary requirements, it would be the question *"How many attendees are vegetarian?"*. Notice how the task input is agnostic to the data since this question could be asked in multiple contexts.

Task Output: This is the answer to the informational task requested by the input. To assess system performance, a metric compares the output to the ground truth averaged over all samples. Common examples of metrics in semi-structured informational tasks are exact match and BLEU [103], which are described in more detail in Chapter 3.

Data structure: This is the underlying representation that the raw data can hold. The data structure determines the type of logical operations to be performed. An example is the difference between representing values in a table as strings, booleans or integers, allowing for different types of comparison. In this thesis, we use two data structures: tables in Chapter 6 and dictionaries (or key-value store) in Chapter 7.

Data T_i and T_{i+1}: This is the raw data pertaining to an instance of the class represented in the form defined by the data structure. Since a tool can transform data multiple times, we use the subscript i to represent the data that the task processor will take as input. At the start of a task, the data will be unmodified, denoted as T_0. Data transformations in this thesis are stateful, meaning transformations modify the underlying representation passed to the next iteration.

Task processor: This component interprets the task input - such as a natural language question pertaining to some data - and a representation of the data at T_i. It then produces two byproducts: first, a representation for the "tool required" decision system, and second, an output to the task. In our dietary requirements example, this component is analogous to a person who does not know how to use a spreadsheet program but attempts to arrive at an answer and communicates the difficulty to a third party. By separating the answer generation from the tool use, we can improve each independently.

The "tool required" decision system: This is a *yes/no* boolean gate used to control the number of iterations in the tool use loop. When the answer is *no*, the loop breaks, yielding the latest answer produced by the task processor as the task output. This decision system can take various task factors or model byproducts into account to determine the need for a tool.

Program generator: This system is responsible for understanding the data transformation needed to help the task processor. It takes the inputs from the task processor and produces an executable program that respects the constraints of the data structure and the available functions. This component needs to have sufficient knowledge of the language syntax so as not to produce non-executable code. This component is a significant object of study in Chapters 4 & 5

The program: This bridges the systems that interpret the task and data (the task processor, "tool required" decision system, and program generator) to the tool that transforms the data (the executable environment). The reason the program acts as a bridge is that the system can simultaneously represent it as raw text and as logical operations when parsed. For instance, language models produce the program as text through auto-regressive generation, which a Python interpreter then parses and executes. This is analogous to the way humans write programs as a sequence of keystrokes that a computer ultimately understands as logical operations.

Program execution: This is where the system applies digital reasoning at scale, solely constrained by the compute resources available, like memory and electricity. This thesis proposes that the properties of this environment are complementary to those of large language models, in that it is precise and input size-agnostic. An example of these properties is CTRL+F on any modern computer; one knows that regardless of a web page's size, the program will yield the correct count. This example also highlights the limitations of these types of digital environments. One can only choose from a limited set of transformations. For instance, one cannot use CTRL+F to find and count synonyms.

Functions and external compute resources: These are the library functions available in the executable environment. These can include system functions native to the programming language as well as library functions. These components are studied in Chapters 6 & 7

External data sources: These represent knowledge that can be made available to the task processor through the executable environment. A good example of such a source is the current date, which can be easily updated, thus patching the frozen parametric knowledge in a language model acting as a task processor. I study this component in Chapter 7.

The framework diagram shown in Figure 2.1 is intended to put the contributions and systems under study in the context of a wider tool use framework. To guide the reader, I will show it with highlighted components at the start of each chapter.

Chapter 3

Background

Before delving into the specific contributions and methodologies that support the thesis statement, this chapter provides the foundational knowledge necessary to understand the core concepts and techniques explored throughout this thesis. We begin by introducing key concepts in neural networks and deep learning, which form the basis of our approach. We then delve into an overview of language models, with a particular focus on transformer architectures and their applications in natural language processing. Following this, we explore the field of semantic parsing, which is crucial for understanding the translation between natural language and code. We also examine various approaches to code translation, retrieval, and conversational state management, which are integral to our research on improving program generation from natural language. Finally, we discuss the metrics commonly used to evaluate the performance of these models in the context of natural language to code translation. This comprehensive background sets the stage for the contributions in combining transformer-based sequence-to-sequence models with retrieval components, leveraging language models for weak supervision, and enhancing tabular question answering via program generation.

All of the work in this thesis is built on top of foundational concepts in machine learning such as deep neural networks, embeddings, attention [4] and gradient based optimisation. However, given the higher level focus of this thesis on using language models as systems readers are referred to foundational texts on deep learning [49,79], optimisation [117] and resources specific to natural language processing [48].

3.1 Language Models

This section details the evolution and core concepts of language models, which are central to the work presented in this thesis. We will trace their development from foundational statistical methods to the large-scale, pre-trained neural architectures that define the current state of the art in Natural Language Processing.

3.1.1 Foundations of Language Modelling

The fundamental goal of language modelling is to assign a probability to a sequence of tokens, a concept with roots in Claude Shannon's information theory [124]. This probabilistic framework is crucial for a wide range of applications, including speech recognition, machine translation, and spelling correction. The probability of a sequence $\mathbf{x}$ is formally calculated using the chain rule of probability:

$$p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i \mid x_1, \ldots, x_{i-1})$$

For decades, the dominant approach to estimating these probabilities was the statistical n-gram model, which simplified the problem by making a Markov assumption: the probability of a token depends only on the preceding $n-1$ tokens. Pioneering work at IBM helped establish these self-organised, statistical models as a cornerstone of mainstream NLP [62].

However, n-gram models faced a critical challenge known as data sparsity. Because most valid word sequences are never observed during training, these models would incorrectly assign a zero probability to unseen n-grams. To address this, a variety of smoothing techniques were developed to redistribute probability mass from seen to unseen events, evolving from simple Laplace smoothing to the highly effective Kneser-Ney smoothing [72]. The quality of these models was typically measured using Perplexity (PPL), which quantifies a model's uncertainty on a test set, with lower scores being better.

Despite their success, a core limitation remained: n-gram models could not generalise based on word similarity (e.g., they treated "cat" and "kitten" as completely unrelated). The breakthrough that overcame this barrier was the introduction of distributed representations in neural language models. A seminal work by Bengio et al. demonstrated that a feed-forward neural network could simultaneously learn continuous vector representations (embeddings) for words and the probability function, vastly outperforming n-gram models by capturing semantic similarity [6]. This marked the beginning of the modern era of language modelling.

3.1.2 Representing Text for Neural Networks

The initial step in processing text with neural networks is converting raw language into a numerical format. This involves two key processes: tokenisation and embedding.

Tokenisation is the process of segmenting input text into a sequence of discrete units, or tokens. While early methods used words or characters, modern systems predominantly use subword tokenisation. This approach balances vocabulary size and sequence length by representing frequent words with a single token and breaking down rare or unknown words into smaller, meaningful subword units. This effectively eliminates the need for an "unknown" token. A common algorithm for this is Byte-Pair Encoding (BPE), a data compression technique [41] adapted for NLP [123], alongside similar methods like WordPiece [144].

Once tokenised, an Embedding Layer maps each discrete token ID to a dense, low-dimensional vector. This layer functions as a learnable lookup table that, through training, captures the semantic and syntactic relationships between tokens. The concept of using dense vectors to represent word meaning was popularised by early models like Word2Vec, which showed that semantic relationships could be captured geometrically in vector space [93].

However, token embeddings alone do not convey sequence order. Non-recurrent architectures like the Transformer process all tokens in parallel and are thus inherently unaware of their position. To solve this, Positional Encodings are added to the token embeddings, injecting information about each token's absolute or relative position in the sequence. The final input representation is a combination of the token's meaning and its position, preparing it for processing by the model's main layers.

3.1.3 Early Neural Approaches for Processing Sequences

Recurrent Neural Networks (RNNs) were specifically designed to process sequential data. They maintain a hidden state, or memory, which is updated at each time step as the model consumes the input sequence one token at a time. A significant advancement in this area was the Long Short-Term Memory (LSTM) network [58]. LSTMs introduced a more sophisticated cell structure with gated mechanisms (input, forget, and output gates) that gave the network finer control over the flow of information into and out of the hidden state. This design was crucial for mitigating the vanishing and exploding gradient problems that plagued simple RNNs, making it possible to learn longer-range dependencies [105]. A similar, slightly simpler gated architecture is the Gated Recurrent Unit (GRU) [24].

Despite these improvements, even LSTMs and GRUs struggled with very long sequences due to their inherently sequential processing bottleneck. A key innovation that addressed this was the attention mechanism, first introduced as an enhancement to RNN-based models [4]. Attention allowed the model, at each step of generating an output, to dynamically create a shortcut to and selectively focus on the most relevant parts of the input sequence, regardless of their distance. This proved to be a powerful precursor to the next major architectural shift, the Transformer, which is covered in the following section.

3.1.4 The Transformer

The Transformer, introduced in the paper "Attention Is All You Need," marked a turning point in sequence modelling by completely replacing recurrence with attention mechanisms [133]. Its core component is the self-attention layer, which allows each token in a sequence to directly attend to all other tokens, calculating a contextually-aware representation in parallel. This parallel processing capability overcame the sequential bottleneck of RNNs, enabling efficient training on unprecedented scales of data.

The original Transformer consists of two main parts:

- **An Encoder Stack**, which processes the entire input sequence to build a rich bidirectional representation for each token. Each token in the input sequence is progressively contextualised through multiple blocks consisting of self-attention and fully-connected layers.
- **A Decoder Stack**, which generates the output sequence one token at a time (auto-regressively). The decoder uses masked self-attention to ensure that predictions for a token can only depend on previously generated tokens. It also employs cross-attention layers, which allow it to focus on the most relevant parts of the encoder's output representations while generating each token.

This powerful and scalable architecture became the foundation for nearly all subsequent large language models [16, 32, 112, 130, 131].

3.1.5 Supervised and Semi-supervised Learning

Before the advent of large-scale pre-training, the dominant paradigm in NLP involved creating specialised neural architectures for each specific task (e.g., a Bi-LSTM with a CRF layer for named-entity recognition). These models were trained from scratch exclusively on labelled, task-specific datasets, a process that was highly data-hungry and did not easily allow for knowledge transfer between tasks.

The paradigm began to shift with the rise of semi-supervised learning. The earliest approaches used unlabelled data to compute word-level statistics that were then used as features in a supervised model [84]. A major step forward was the development of pre-trained word embeddings like Word2Vec [93] and GloVe [108]. These models were trained on vast amounts of unlabelled text to learn a single, static vector representation for each word in the vocabulary. These pre-trained embeddings were then used to initialise the first layer of a task-specific neural network. This injected broad semantic knowledge into the model from the start, leading to better performance and faster training, especially with limited labelled data. This approach, however, still trained the bulk of the network from scratch for each task.

3.1.6 Unsupervised Pre-training

The major breakthrough that defines modern NLP was the idea of pre-training the *entire* deep neural network, not just the embedding layer, on a general, unsupervised objective. This approach leverages vast amounts of unlabelled text from the internet to learn universal linguistic patterns, which can then be adapted to specific tasks.

An early success demonstrating this potential was ELMo, which used a pre-trained bidirectional LSTM to generate deep, context-sensitive word embeddings that changed based on the sentence they appeared in [109]. The scalability of the Transformer architecture, however, enabled pre-training at a much larger scale, leading to two dominant paradigms:

- **Auto-regressive language modelling**: The model is trained to predict the next token in a sequence. This is the objective used by models in the GPT family [112].
- **Masked language modelling (MLM)**: The model is trained to predict randomly masked tokens in the input, allowing it to learn deep bidirectional context. This objective was famously used by BERT [32].

A more general form of this is denoising auto-encoding, where the model learns to reconstruct the original text from a corrupted version. This includes the MLM objective used by BART [81] and the span corruption objective used by T5, where the model must restore entire missing spans of text [114].

The success of this paradigm is fundamentally dependent on web-scale data, often sourced from corpora like Common Crawl. Extensive filtering is required to create high-quality training sets, such as the C4 (Colossal Clean Crawled Corpus) developed for T5 [114]. The effectiveness of this approach is explained by Scaling Laws, which empirically show that model performance scales predictably with increases in model size, dataset size, and training compute [66]. This scaling can unlock Emergent Abilities—capabilities like multi-step reasoning that are not present in smaller models but appear spontaneously in larger ones [136]. The "Chinchilla" study further refined these laws, demonstrating that for optimal performance, model and dataset size should be scaled in proportion, revealing that many prior models were "undertrained" [59].

3.1.7 Adapting Pre-trained Models to New Tasks

Once pre-trained, a model's general-purpose knowledge is adapted to solve specific problems through a process known as transfer learning. Several strategies exist for this adaptation.

The most common method is supervised fine-tuning, where all of the pre-trained model's parameters are updated on a smaller, labelled dataset for the target task. For highly specialised domains, this can be preceded by domain-adaptive pre-training, where the model is first continually pre-trained on in-domain text. This has been highly effective for tasks like tabular question answering (TQA), with models like TaBERT [151] and TaPas [57] learning representations grounded in both text and tables. Later models such as UnifiedSKG [145] and OmniTab [64] advanced this by framing diverse structured data tasks within a unified text-to-text format.

A more generalist approach is instruction tuning, which fine-tunes a model on a massive and diverse collection of tasks formatted as natural language instructions. This improves the model's ability to follow commands and generalise to unseen tasks in a zero-shot setting, a concept pioneered by models like T5 [114] and solidified by FLAN [135]. Research has shown that instruction-following ability scales with the number of tasks, model size, and the inclusion of reasoning data [25].

Finally, an important emergent capability of very large language models is in-context learning (ICL), which allows for adaptation at inference time without any weight updates. This is achieved by crafting an input prompt that includes a few examples (demonstrations) of the task alongside the query, a capability famously demonstrated by GPT-3 [16]. The effectiveness of ICL is highly sensitive to the prompt's design, a practice known as prompt engineering. A critical innovation in this area is chain-of-thought (CoT) prompting, which significantly boosts reasoning abilities by including intermediate reasoning steps in the prompt's examples [137]. This teaches the model to "think step by step," significantly improving accuracy on complex tasks. A simplified variant, zero-shot CoT, shows that simply appending a phrase like "Let's think step by step" can trigger the model's latent reasoning abilities without needing any examples [73].

3.2 Search Methods

Beyond the direct application and fine-tuning of language models themselves, another set of techniques crucial to several contributions in this thesis involves retrieving relevant information to augment language model reasoning or provide context for generation. Therefore, in this section, we outline the various information retrieval methods employed. This includes lexical methods like BM25 and neural methods like MonoBERT and DuoBERT.

3.2.1 Lexical methods

Lexical search methods rely on the co-occurrence of terms between a query and a document to find the most relevant documents within a corpus. A score is derived from the number of terms that overlap using a weighting that ensures the most relevant terms in a query and document influence the score. A common weighting for terms involves collecting the frequency of a term in a document (TF or term frequency), normalised by the frequency of that term within the corpus (DF or document frequency). This is commonly expressed in information retrieval literature as TF-IDF, where we multiply TF by the inverse of the document frequency (IDF) [127]. BM25 [65] is a widely used implementation of this search method, which includes additional improvements such as stemming and stop-word removal to increase the likelihood of semantically equivalent term matches. This search method is frequently used in this thesis as a first-pass method for finding documents from a corpus given a search query.

3.2.2 Neural methods

Neural search methods leverage deep learning architectures to capture semantic relationships between queries and documents. Approaches based on Siamese networks [15] involve encoding the document and query into a common embedding space, allowing for nearest-neighbour algorithms to scalably order results by similarity [67]. Although slower, other models like

MonoBERT [97] and DuoBERT [99] utilise transformer-based architectures, BERT in particular, to jointly encode documents and the query to derive a scalar value used in downstream document ranking. MonoBERT employs a single BERT encoder to score query-document pairs, while DuoBERT takes a query and two documents as input to produce a relative relevance score between the documents. Given the required joint interaction between query and document(s) for these methods, they are typically more expensive and thus reserved for re-ranking a pre-existing pool of documents, such as those obtained from BM25.

3.3 Approaches for Code Translation

Having covered foundational concepts in language modelling and complementary search techniques, we now turn to a central task directly addressed by this thesis: the translation of natural language into code. In this section, we cover various methods specifically developed for this purpose, setting the stage for the novel contributions presented in later chapters. Given the structured nature of code, many approaches are tuned to preserve syntactic correctness.

3.3.1 Semantic Parsing

Semantic parsing can be viewed as a foundational attempt to bridge the gap between ambiguous natural language input and formal, executable meaning representations. Semantic parsing is the task of translating natural language utterances into formal meaning representations, such as logical forms, database queries, or executable code [154]. This field has gained significant attention in natural language processing due to its applications in question answering, task-oriented dialogue systems, and code generation [7].

The process of semantic parsing typically involves several key steps:

1. Lexical analysis: Identifying and categorising the words and phrases in the input utterance.
2. Syntactic parsing: Determining the grammatical structure of the input.
3. Semantic interpretation: Mapping the syntactic structure to a semantic representation.
4. Logical form generation: Constructing a formal representation of the input's meaning.

Various approaches have been developed for semantic parsing, including:

1. Rule-based methods: These use hand-crafted rules to map natural language to formal representations [143]. Early rule-based systems primarily relied on hand-crafted grammars and logical rules.

2. Statistical methods: These learn probabilistic models from annotated data to perform the parsing [155]. These introduced probabilistic elements but still often relied on significant feature engineering related to syntactic structure.

3. Neural network-based methods: These leverage deep learning techniques to learn the mapping end-to-end [33,150]. More recent neural methods, particularly sequence-to-sequence models, leverage the pattern recognition abilities of deep learning to learn the complex mapping from the nuances of natural language utterances directly to the target formal representation (e.g., SQL, Python code), often in an end-to-end fashion.

Recent advancements in semantic parsing have focused on improving generalisation to unseen domains [56], handling complex and compositional queries [106], and reducing the need for large amounts of annotated training data through techniques like transfer learning and few-shot learning [126].

Semantic parsing plays a crucial role in bridging the gap between natural language and formal representations, making it an essential component in many natural language understanding tasks and applications.

3.3.2 Tree-structured methods

Within the broader field of semantic parsing, and particularly relevant for the task of code translation due to the inherent structure of programming languages, tree-structured methods play a significant role. These methods for code translation leverage the hierarchical nature of programming languages, typically operating on Abstract Syntax Trees (ASTs). These approaches aim to improve translation accuracy and coherence by preserving code structure.

Key methods include Tree-to-Tree Neural Networks [22], which map between source and target language ASTs, and Abstract Syntax Networks (ASN) [111], combining sequence-to-sequence models with grammar models for syntactically valid code generation. Recent advancements incorporate Graph Neural Networks (GNNs) [2] to process code as graphs, allowing for richer representations of structure and dependencies.

Tree-structured methods offer advantages such as improved handling of long-range dependencies and better generalisation to unseen patterns. However, they require more complex model architectures compared to sequence-based approaches.

3.3.3 Approaches to code retrieval

Alongside methods that generate code from scratch, another important strategy for obtaining relevant code from natural language descriptions is retrieval based on search techniques covered previously. Code retrieval approaches aim to find existing code snippets from a repository that match a given query and can be broadly categorised into several types. Information retrieval-based methods adapt traditional techniques like TF-IDF or BM25 to rank code snippets based on query relevance, but face limitations in semantic understanding and handling code-specific elements. Neural network-based approaches leverage deep learning advancements, employing techniques such as Siamese networks for joint embeddings of natural language and code [51], and transformer-based models to capture semantic relationships [36]. Hybrid approaches combine traditional information retrieval methods with neural networks to harness the strengths of both paradigms. Semantic parsing-based methods aim to understand query intent and map it to formal representations for more effective code repository searches. Lastly, code-specific retrieval techniques consider the unique structure and properties of code, such as ASTs or control flow graphs, to enhance retrieval accuracy [61].

3.4 Tabular Question Answering

The ability to translate natural language to code, whether through generation or retrieval, forms a prerequisite for a more advanced capability central to this thesis: enabling systems to use code as a tool. Tool use, particularly over structured data, naturally extends from semantic parsing, serving as an intermediate way for systems to interact with and manipulate such data. Question answering is often framed as an overarching task where a model must implicitly determine the intermediate steps required for a solution. Such approaches typically require strong supervision [34, 149] or reinforcement learning [156], assume coherently formatted data, and, when using semantic parsing, depend on an expressive target language.

Alternative approaches learn a joint table-question-answer mapping. Sequence-to-sequence models [129] execute and simulate formal programs [153]. Intermediate executable modules were integrated [96], while transformer-based models [81, 114, 133] leveraged unsupervised language capabilities [57, 64, 145, 151].

BART [81], built on the Transformer architecture [133], employs denoising pre-training. It processes tabular question answering tasks by combining queries and linearised tables within a 1024 token window using special tokens: $x = q\,\texttt{[HEAD]}, c1, ..., cN, \texttt{[ROW]}, 1, r1, \texttt{[ROW]}, 2, r2$.

TapEx [88] extends BART through SQL execution training on synthetic data, achieving leading results on WikiSQL's weak supervision benchmark.

OmniTab [64] builds upon TapEx with natural language fine-tuning, converting SQL to questions and leveraging table-aligned text. It leads WikiTableQuestions performance through specialised training.

UnifiedSKG [145] adapts T5 [114] for structured knowledge tasks using a unified text-to-text approach across diverse data types (tables, knowledge bases, semantic parsing, etc.).

FlanT5 [135] balances specialisation and generality through instruction-tuning across 62 NLP tasks, establishing strong tabular question answering performance via in-context learning.

GPT-3 [16] demonstrates unsupervised tabular question answering capabilities through zero-shot chain-of-thought reasoning [21, 73, 137], providing step-by-step solutions.

3.5 Reasoning with Language Models

Execution-loop models arise from language models' ability to explain reasoning [137], improving compositional questions [157] and symbolic manipulation [17, 100, 142]. Tabular question answering language models generate chains of thought with sub-question answers [21].

Subsequent advances in code-focused language models have led to an interest in combining question decomposition and program interaction [20]. Many recent approaches, like Toolformer [119], Program Assisted Learning (PAL) [42], and tool-augmented language models (TALM) [104], attempt to integrate the language model's reasoning with external tools (like calculators or search APIs). They typically involve the language model deciding when to call a predefined tool and interpreting its output. Our approach differs by focusing on the language model generating a task-specific program, specifically a filter, tailored to the input data, rather than just calling a fixed external API. This allows for more flexible and potentially more powerful interaction with semi-structured data. These methods interleave execution and natural language reasoning but face capacity limitations due to context length requirements.

3.6 Approaches to Conversational State Management

This thesis explores how augmenting language models through program generation can enhance their performance when interacting with various forms of structured data. Long, task-oriented conversations represent a dynamic instance of such structured data, where an evolving dialogue state and a sequence of required steps must be coherently managed. The practical application of this thesis's core argument, that language models generating and using code as tools, is prominently demonstrated in Chapter 7 within the context of the GRILLBot system, which combined fluent conversation with the management of long, structured tasks during the Amazon Alexa Prize TaskBot competition. To fully appreciate the challenges GRILLBot addressed and how its code-generating Neural Decision Parser provided an effective solution, it is essential to first understand the established landscape of conversational state management. Thus, this section reviews various approaches providing the necessary context for the innovations presented later.

3.6.1 Rule-based approaches

Early dialogue systems employed Finite State Machines (FSMs), modelling conversations as directed graphs where each node represents a system prompt and each edge a user response triggering state transitions [12]. FSMs offered high control and predictability, but were brittle to deviations from the scripted paths, making them impractical for open-ended or mixed-initiative dialogues.

Frame-based models generalised FSMs by representing tasks as sets of slots to be filled, enabling partial information updates and flexible slot order [10]. This approach, still widely used in production systems, supports mixed initiative and scales well to new domains by modifying the slot ontology [152]. However, frame-based systems rely heavily on developer-specified ontologies and struggle with unanticipated user inputs.

Plan-based dialogue managers further extended rule-based methods by modelling dialogues as hierarchical plans decomposed into sub-goals. Frameworks such as RavenClaw [12] and TRIPS [37] exemplify this approach, allowing task interleaving, clarification sub-dialogues, and repair strategies. While powerful, these systems required substantial knowledge engineering to specify plan hierarchies and update rules.

3.6.2 Statistical and probabilistic methods

To address brittleness and uncertainty inherent in rule-based systems, statistical methods introduced explicit modelling of uncertainty via Partially Observable Markov Decision Processes (POMDPs) [141]. POMDPs maintain a probabilistic belief state over possible dialogue states, updating it with Bayesian inference at each turn based on observed user input and system actions [152]. This formulation enables principled optimisation of dialogue policies via reinforcement learning to maximise long-term task success under uncertainty.

Despite strong theoretical foundations, pure POMDP approaches remain computationally challenging for large state spaces, leading to approximate methods such as summary-state POMDPs [152]. Practical systems often adopt hybrid statistical architectures, combining rule-based modules with probabilistic state trackers to handle ambiguity in slot filling [128].

The Dialogue State Tracking Challenge (DSTC) series [140] catalysed advances in statistical dialogue state tracking, establishing benchmarks for evaluating belief tracking accuracy and driving progress toward data-driven models that generalised across domains.

3.6.3 Neural approaches

The advent of deep learning brought end-to-end neural models for dialogue state tracking and policy optimisation. Neural Belief Trackers (NBT) [95] replaced manual semantic lexicons with distributed word embeddings, allowing generalisation across synonyms and paraphrases without extensive manual feature engineering.

Subsequent models like Sequicity [80] and UBAR [147] unified dialogue state tracking, policy prediction, and response generation into single sequence-to-sequence models. These systems leveraged recurrent and transformer-based architectures to implicitly model dialogue context, allowing fully data-driven training from annotated dialogues.

Pre-trained language models (PLMs) such as BERT and GPT-2 substantially improved language understanding in dialogue pipelines. BERT-based trackers like SOM-DST [71] and TripPy [54] demonstrated state-of-the-art slot tracking performance by encoding full dialogue history via transformers. Sequence generation models like SimpleTOD [60] used GPT-2 to jointly generate dialogue states and responses, leveraging large-scale language pre-training for strong few-shot performance.

However, fully end-to-end neural systems face challenges in interpretability, consistency, and data efficiency. Implicit latent state representations make debugging difficult, and maintaining dialogue consistency over long interactions remains problematic [115].

3.6.4 Hybrid neural-symbolic systems

Recognising the limitations of purely neural or purely symbolic systems, hybrid approaches have emerged to combine the strengths of both. Hybrid Code Networks (HCN) [139] integrate recurrent neural networks for policy learning with explicit domain-specific code modules for handling structured actions, reducing data requirements while preserving interpretability.

Production systems such as Rasa [11] and Alexa Conversations [1] similarly blend rule-based schemas, API specifications, and learned policies. Alexa Conversations employs a dialogue simulator to generate diverse synthetic training dialogues from a developer-specified schema of APIs, slots, and action templates, enabling scalable policy learning from limited seed examples.

These hybrid architectures reflect a growing consensus that structured intermediate representations (e.g., slots, dialogue acts, API calls) serve as useful inductive biases, enabling efficient learning, verification, and integration with external systems while allowing neural models to handle linguistic variability.

3.6.5 Programmatic and code-based dialogue policies

Recent work has advanced the paradigm of treating dialogue policies themselves as code generation tasks. Instead of predicting dialogue acts or slot values, models generate executable programs in domain-specific languages (DSLs) that precisely specify the next system action. Amazon's Dialogue Markup Language (DML) [1] formalises API calls as programmatic actions for dialogue execution.

Hybrid Code Networks [139] introduced early versions of this concept by allowing RNNs to output tokens that directly invoked external functions. Later neuro-symbolic approaches extended this further by generating full programs or call graphs representing dialogue policies [83].

The ability to generate structured, executable policies offers strong advantages: model outputs are interpretable, verifiable, and amenable to constraint enforcement; domain-specific knowledge can be embedded in external functions; and tool use capabilities (e.g., invoking calculators, databases, web services) become naturally integrated within dialogue flows. This paradigm aligns with broader trends in neuro-symbolic reasoning, retrieval-augmented language models, and tool-augmented language models.

The Neural Decision Parser (NDP), discussed in Chapter 7, builds directly on this direction, tasking the language model to generate Python code that fully specifies system behaviour at each turn. The modified structured representation is then fed back into the NDP on the next turn to contextualise the incoming utterance.

3.7 Metrics Used

In this section, I describe metrics commonly used throughout this work in the context of natural language to code translation and question answering.

Exact Match

The *exact match* metric evaluates the accuracy of a generated text by checking if it exactly corresponds to a reference text, without any deviations. This metric is defined as a binary indicator, where a value of 1 indicates a perfect match and 0 indicates a mismatch:

$$\mathrm{EM}(\mathbf{x},\mathbf{x}') = \begin{cases} 1 & \text{if } \mathbf{x} = \mathbf{x}', \\ 0 & \text{otherwise.} \end{cases}$$

This metric is particularly stringent as it does not tolerate any errors in the generation, including rephrasings, synonyms, or even minor grammatical discrepancies. While useful for tasks that require high precision, such as code generation or factual reporting, it does not fully capture the quality in more subjective text generation tasks, where paraphrasing could still preserve the intended meaning.

BLEU

BLEU is a metric for automatic evaluation of sequence similarity originally proposed for machine translation [103]. Given a reference summary $\mathbf{x} = x_1,\ldots,x_l$ and a candidate summary $\mathbf{y} = y_1,\ldots,y_m$, the BLEU score involves two arguments: a candidate string $\hat{y}$ and a list of reference strings $(y^{(1)},\ldots,y^{(N)})$. The idea is that $\mathrm{BLEU}(\hat{y};y^{(1)},\ldots,y^{(N)})$ should be close to 1 when $\hat{y}$ is similar to any of $(y^{(1)},\ldots,y^{(N)})$, and close to 0 if not. This measure is analogous to a language teacher scoring the quality of a student's translation $\hat{y}$ by comparing how closely it follows the reference answers $(y^{(1)},\ldots,y^{(N)})$.

The modified n-gram precision is a generalisation of the basic case with one candidate sentence and one reference sentence. For a single reference sentence, it is defined as follows:

$$p_n(\{\hat{y}\};\{y\}) = \frac{\sum_{s\in G_n(\hat{y})} \min(C(s,\hat{y}),C(s,y))}{\sum_{s\in G_n(\hat{y})} C(s,\hat{y})}$$

where $G_n(\hat{y})$ denotes the set of n-grams in $\hat{y}$, and $C(s,\hat{y})$ and $C(s,y)$ denote the count of n-gram s in $\hat{y}$ and y, respectively. In the context of this thesis, we use BLEU with only a single reference sentence constrained by the availability of multi-reference data.

One of the pitfalls of using BLEU as a sequence similarity metric is that it is reliant on the presence of an n-gram exact match and not semantic similarity. We refer to other works for details on evaluation for semantic similarity [122].

Recall

Recall, particularly in the context of information retrieval, measures the ability of a system to retrieve all relevant documents from a collection. Recall at rank k (Recall@k) specifically evaluates the fraction of the total number of relevant documents that are present within the top k retrieved results. It is a measure of completeness and is defined as:

$$\text{Recall@k} = \frac{|\{\text{relevant documents}\} \cap \{\text{top } k \text{ retrieved documents}\}|}{|\{\text{total relevant documents}\}|}$$

A high Recall@k indicates that the system is effective at finding most of the relevant items within a reasonably short ranked list.

Mean Average Precision

Mean Average Precision (MAP) is a popular metric for evaluating the performance of ranked retrieval systems. It provides a single-figure measure that accounts for both precision and recall. To understand MAP, we first define Average Precision (AP) for a single query. AP is the average of the precision values obtained at each rank that contains a relevant document.

$$\text{AP} = \frac{\sum_{k=1}^{N}(P(k) \times \text{rel}(k))}{|\{\text{relevant documents}\}|}$$

Here, N is the total number of retrieved documents, $P(k)$ is the precision at rank k, and $\text{rel}(k)$ is an indicator function that is 1 if the document at rank k is relevant and 0 otherwise. AP heavily rewards systems that place relevant documents at the top of the ranked list.

MAP is then simply the mean of the Average Precision scores calculated over a set of queries Q:

$$\text{MAP} = \frac{1}{|Q|}\sum_{q=1}^{|Q|} \text{AP}(q)$$

Normalised Discounted Cumulative Gain

Normalised Discounted Cumulative Gain (NDCG) is a measure of ranking quality that is particularly useful when documents have varying degrees of relevance (i.e., graded relevance scores) [29]. It is based on two principles: highly relevant documents are more valuable than marginally relevant ones, and relevant documents are more valuable when they appear earlier in the results.

The metric starts with Cumulative Gain at rank k (CG@k), which is the sum of the relevance scores of the top k results:

$$\mathrm{CG@k} = \sum_{i=1}^{k} \mathrm{rel}_i$$

To account for the position, Discounted Cumulative Gain (DCG@k) applies a logarithmic penalty to documents at lower ranks:

$$\mathrm{DCG@k} = \sum_{i=1}^{k} \frac{\mathrm{rel}_i}{\log_2(i+1)}$$

Finally, to make the score comparable across different queries (which may have different numbers of relevant documents), DCG is normalised by the Ideal Discounted Cumulative Gain (IDCG@k), which is the DCG score of a perfect ranking. This gives the final NDCG@k score, which ranges from 0 to 1.

$$\mathrm{NDCG@k} = \frac{\mathrm{DCG@k}}{\mathrm{IDCG@k}}$$

3.8 Conclusion

This background section has provided a comprehensive overview of key concepts, methodologies, and approaches foundational to my thesis on improving natural language conditioned program generation. We have covered neural networks and deep learning fundamentals, language models, semantic parsing, code translation methods, code retrieval approaches, conversational state management, and evaluation metrics. This extensive background contextualises our research, positioning us to explore the superiority of transformer-based sequence-to-sequence models in program generation, the integration of retrieval components, the use of general-purpose language models for weak-supervision data, the advantages of language infilling objectives over traditional translation objectives, and the application of these findings to tabular question answering. The subsequent chapters will build upon this foundation to present our methodologies, experiments, and results, validating the claims put forth in my thesis statement.

Chapter 4

Natural language to code translation

This work was carried out in 2019–2020 and published at SIGIR 2020 [47]. At that time, natural language to code translation was dominated by AST-based models such as Abstract Syntax Networks [111] and LSTM sequence-to-sequence systems, and large-scale pre-trained models for code did not yet exist. The Transformer [133] had demonstrated strong performance in machine translation but had not been applied to code generation. AST-based models operate only at the generation stage and cannot represent natural language and code within a single unified framework, which limits their applicability to the full tool use loop the thesis develops. This chapter shows that the Transformer outperforms other general-purpose, syntax-agnostic architectures while also validating the use of a fixed tool.

With the foundational background concepts in place, this chapter begins to address the core contributions of the thesis, focusing first on the challenge of improving natural language to code translation (Hypothesis 1). At the time of writing, many of the works in this field involved specialised architectures that are specifically and solely built for generating code [111].

This chapter serves two purposes to validate the thesis statement: First, it depicts the first use of a syntax-agnostic architecture, the Transformer, to generate programs and compares it with other general-purpose methods like retrieval. Second, it shows that a retrieval system - external to the neural translation component - can act as a fixed tool to further improve code translation accuracy. To illustrate the two perspectives from which to view the contributions in this chapter, Figures 4.1 and 4.2 show how they fit within the tool use framework from Chapter 2.

As shown in Figure 4.1, writing code is a key requirement for a system to use executable programs as a means to improve a downstream task, thus acting as a program generator. This chapter provides an initial sentence-level benchmark for the ability of general-purpose architectures to semantically map between English and code, directly addressing Hypothesis 1.1. Before language models can effectively use programs as tools, they must be able to reliably generate them from natural language specifications. This is the macro perspective of this chapter.

Figure 4.2 prompts the reader to view program generation as an instance of a general sequence-to-sequence task. A paired corpus of natural language and its equivalent representation in code

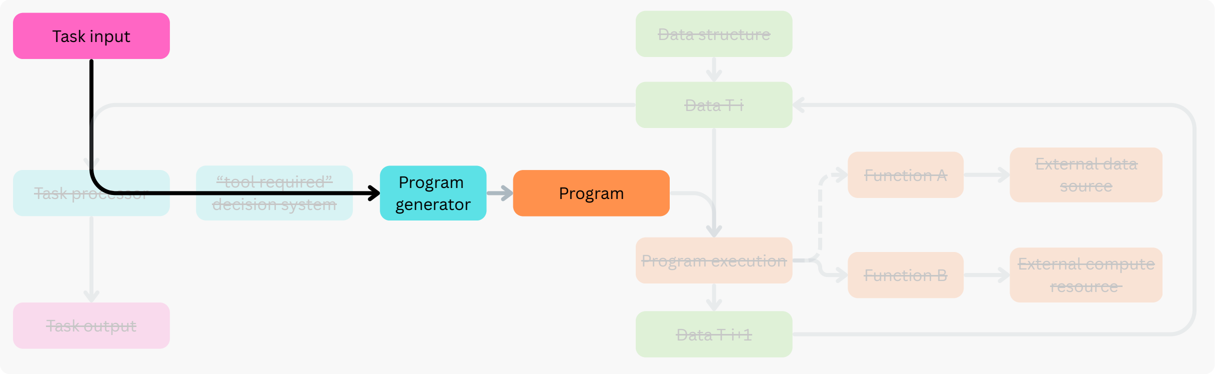


Figure 4.1: The tool use framework highlighting the components that are studied in this chapter from a macro point of view. The overall task of translating natural language to code is that of the program generator. **The input to the program generator is an English statement represented as text.** **There are no decision-making components in this system and thus the program generator is the sole focus within the framework.** **The target output for the program generator is a program that is semantically equivalent to the input. However, the program is not executed.** **Since the program generator is the only component studied under this macro perspective and no execution, there are no data structures to modify.**

acts as a semi-structured data source. This corpus can be queried by executing a search. Finally, the top results of the search can be provided as input to the task processor to improve the mapping from input to output. Although the query used for the retrieval system is an argument that is specific to each task in the dataset, we consider a retrieval system a fixed tool since it is not composed with any other functions. This is the micro perspective of this chapter.

4.1 Introduction

In my thesis statement (Section 1.6), I postulate that sequence-to-sequence models are capable of generating programs and using them as tools to improve downstream question-answering task performance. To break this hypothesis down, in this chapter, we study the modelling component of this statement: *generating programs according to natural language descriptions*. Specifically, we study the capabilities of the Transformer architecture for English to code translation in the context of other common approaches like LSTMs and code retrieval. Thus, we directly address Hypothesis 1.1. Following this, we address Hypothesis 1.2 by studying the effect that combining retrieval with the Transformer has on English-to-code translation.

Natural language and programs often co-occur as software is built for specific purposes resulting from an overall project description, as seen in Figure 4.3. The type of natural language that can be associated with programs can vary in specificity according to the granularity of the associated code. `README.md` files and project proposals can contain high-level requirements pertaining to an entire codebase. In contrast, in-line comments can focus on single-line instructions explaining the reason for the existence of a single statement within a software project.

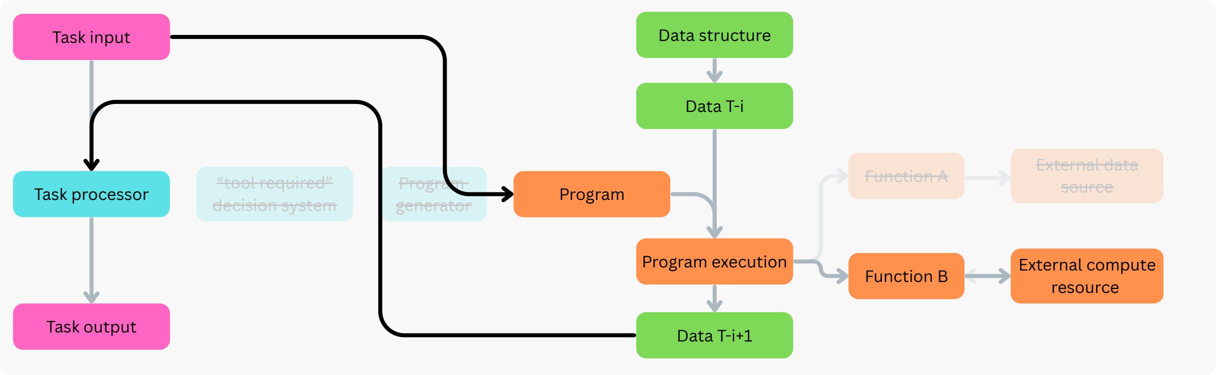


Figure 4.2: The tool use framework showing the micro perspective of this chapter, where we consider translating English to code as an instance of any sequence transduction task. **The task input is an English sentence, and the output is a semantically equivalent sequence of code. A large corpus of equivalent English and code serves as the semi-structured data source in the framework. The task input is used as a query in a retrieval system to surface the most similar pairs in the corpus. This constitutes a fixed tool that is always executed. The task processor is fed the task input and the top retrieved results using both to produce the task output.**

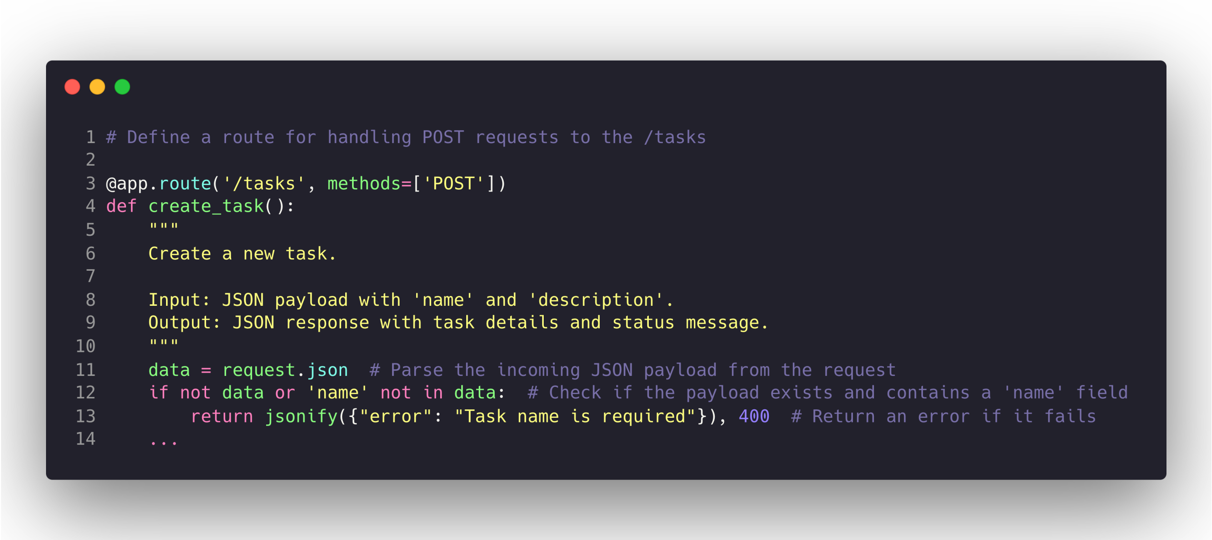
Furthermore, a single line of code may refer to more code called as a result, such as *functions* and *API calls*.

For the purpose of tool use, the right level of granularity to generate code is at the single statement level. This granularity typically corresponds to the execution of specific actions or functions. Single-line statements can map to natural language descriptions, which can facilitate the creation of more accurate natural language to code models.

In this chapter, we focus on generating single-line statements from single sentences of natural language descriptions. The ability to translate natural language into code requires both an understanding of the syntax and semantics of the programming language, as well as an understanding of the intent expressed in the natural language description. This process can often involve ambiguity, as the same natural language description can often correspond to multiple valid code snippets, each with subtle differences in functionality or structure.

A programmer must possess a broad understanding of various programming languages, libraries, and methodologies to write code efficiently. However, the vast amount of structured information needed to accomplish this can be overwhelming, leading to a reliance on online resources for examples or syntax explanations. This reliance can prolong the development process and negatively impact productivity.

Code retrieval [94] can be a valuable tool for programmers. However, it often lacks the flexibility to accommodate the diverse needs of programmers and struggles to adjust to specific contexts. In contrast, code generation aims to address these limitations by enabling programmers to describe their ideas in natural language, which are then translated into code through an algorithmic process. This approach allows programmers to concentrate on more abstract tasks.



```
 1 # Define a route for handling POST requests to the /tasks
 2
 3 @app.route('/tasks', methods=['POST'])
 4 def create_task():
 5     """
 6     Create a new task.
 7
 8     Input: JSON payload with 'name' and 'description'.
 9     Output: JSON response with task details and status message.
10     """
11     data = request.json  # Parse the incoming JSON payload from the request
12     if not data or 'name' not in data:  # Check if the payload exists and contains a 'name' field
13         return jsonify({"error": "Task name is required"}), 400  # Return an error if it fails
14     ...
```

Figure 4.3: Source code showcasing various levels of English descriptions. Line 1 shows a high-level specification description. Lines 6 to 9 are a function-level description. Each of lines 11, 12, and 13 contains line-level comments, which are the level of granularity we use in this chapter.

In the field of Neural Machine Translation (NMT) for code generation, traditional approaches have relied on RNN-based encoder-decoder architectures, with Long Short-Term Memory (LSTM) networks being a popular choice. Although RNN-based models have proven effective in various translation tasks [4, 129], more recent advancements like the Transformer model [133] have demonstrated substantial improvements in NMT performance, largely due to their self-attentive design. Nevertheless, a common limitation shared by these architectures is their inability to effectively integrate external knowledge sources.

This chapter introduces the Relevance Transformer, a novel model that leverages pseudo-relevance feedback during the translation decoding process. Building upon the techniques outlined in [76], the model introduces a positive bias to enhance the quality of auto-regressive generation. This bias is generated by identifying and retrieving code snippets relevant to the given English description, then extracting common tokens in proportion to their relevance to the model. Evaluations conducted on standard benchmark datasets demonstrate consistent improvements over both retrieval and generation baselines. Notably, significant performance gains are observed on the CoNaLa dataset [148], a realistic collection derived from Stack Overflow questions.

4.2 Approaches to Code Generation and Retrieval

4.2.1 Problem Formulation

In the context of natural language to code translation, we can define our objective as follows: given a natural language query q, our aim is to produce the most appropriate code snippet c that fulfils the requirements specified in the query.

We can formally express this task as:
Input: The query q is tokenized into a sequence $\{q_i\}_{i\in[0,\ldots,n]}$, where i represents the token's position in the sequence.
Output: The code snippet c is tokenised into a sequence $\{c_i\}_{i\in[0,\ldots,m]}$.

It is important to note that c may be obtained through two distinct methods: retrieval from existing code repositories or generation using a model. The resulting output typically consists of a concise code snippet, often equivalent to one or a few lines of code.

4.2.2 Baseline Retrieval

A crucial element of our model architecture is the retrieval mechanism. This component is tasked with ranking relevant documents based on their similarity to an input query. In our scenario, the query corresponds to the natural language description from the code-description pair, denoted as (q,c). Our search space encompasses all English descriptions within the training dataset. The retrieval algorithm assigns a score to each document d using a similarity function $RS(q,d)$. Through experimentation, we have identified two effective approaches for snippet retrieval. The first utilises a BM25 implementation via Lucene, accessed through the PyLucene interface. The second employs the similarity scoring function from the ReCode system [53], which operates at the token level. After evaluating both methods, we have chosen to proceed with BM25 due to its superior computational efficiency.

The retrieval algorithm's ranking is utilised to select the top k documents. The code is then extracted from these document pairs and employed in one of two ways: either as the final output for our baseline retrieval methods, or as a guiding mechanism for our proposed Relevance model.

4.2.3 Baseline Transformer

At the core of our system lies the Transformer architecture [133]. This model employs multiple self-attention layers within an encoder-decoder framework, enabling the mapping of variable-length input sequences to variable-length output sequences. The output generation process is auto-regressive, producing a conditional distribution over the entire vocabulary at each time step t. During the training phase, the model implements a look-ahead attention mask to conceal

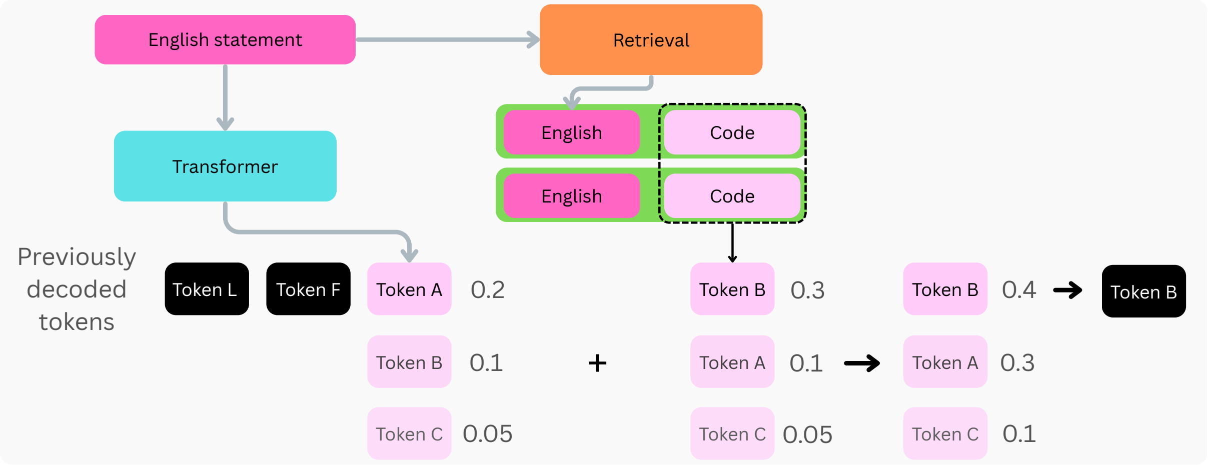


Figure 4.4: High-level diagram showing how retrieval is combined with the transformer model to produce the output sequence. Both the Transformer and the retrieval stages produce probability distributions, which are interpolated, yielding a new probability distribution.

future predictions from the current step. This ensures that predictions are based solely on the English tokens q and the output sequence generated up to that point, $c_{0:t-1}$.

Considering the relatively smaller scale of our datasets compared to those typically used with transformers, we have adapted the model's architecture accordingly. Our implementation features two attention layers for both the encoder and decoder, four attention heads, an embedding dimension of 512, and a pointwise feed-forward network dimension of 1024.

4.2.4 Relevance Transformer

This section provides a comprehensive description of the Relevance Transformer, our proposed model designed to address the specific challenges associated with code generation. It is important to note that preliminary approaches, such as the simple concatenation of top-ranked code results to the input, were found to be ineffective. Relevance Transformer incorporates two primary components that contribute to significant improvements over the baseline implementation: pseudo-relevance feedback decoding and input token copying. These components will be discussed in detail in the following subsections.

Pseudo-Relevance Feedback

A key feature of our proposed network is the implementation of a sequence-aware pseudo-relevance feedback decoding method, building upon the work of [76]. As seen in Figure 4.4, the retrieval component acts as a bridge, providing the generative model (the Transformer) with concrete examples of relevant code snippets. This pseudo-relevance feedback helps ground the generation process, biasing it towards plausible and syntactically common code structures related to the input query. During each decoding step, our copy-augmented Transformer generates a probability distribution over all tokens in the vocabulary, including positional out-of-vocabulary terms. We denote this distribution as $M(q, c_{0:t-1})$, where $c_{0:t-1} = \{c_0, ..., c_{t-1}\}$ represents the current decoded sequence. To enhance the quality of decoding, we retrieve the top k documents $D(q,k)$ and emphasise a set of common words $ST(n)$ found in these results. This is achieved through the interpolation of normalised token frequency scores with the original NMT distribution, as described in Equation 4.1.

$$\begin{aligned} P(w_t|q, c_{0:t-1}) =& [\lambda \cdot M(q, c_{0:t-1}) + (1-\lambda) \cdot RF(q, w_t) \\ & \cdot RP(c_{0:t-1}, w_t)] \cdot Z \end{aligned} \tag{4.1}$$

$$\begin{aligned} & fr(w_t, d) = count(w_t, d)/length(d) \\ & RF(q, w_t) = \left[1 - \nVdash_{ST(n)}(w_t)\right] \cdot \sum_{d \in D(q,k)} fr(w_t, d) \cdot RS(q, d) \end{aligned} \tag{4.2}$$

In this equation, Z represents the normalisation constant. The model considers both the score assigned by the retrieval algorithm and the document length for each token, with the aim of emphasising tokens from highly-ranked snippets. Although there is no certainty that a highly-ranked snippet will invariably provide appropriate word suggestions for the output, the aggregation of multiple top-scoring snippets increases the confidence in elevating the probability of commonly occurring words, as demonstrated in Equation 4.2.

Furthermore, the model accounts for terms that have already appeared in the current decoded sequence. To address this, a repetition penalty is implemented (Equation 4.3), which modulates the probability assigned to a term based on its prior occurrence in the prediction. This mechanism helps to maintain diversity in the generated output and prevent excessive repetition of terms.

$$RP(c_{0:t-1}, w_t) = [1 - \nVdash_{c_{0:t-1}}(w_t)] \tag{4.3}$$

Copy Generation Methods

The concept of copy methods originates from Pointer Networks [134], which utilise the attention distribution over the input sequence to select an element from the input at each decoding step. While Pointer Networks are limited to copying elements from the input, Copy Generator Networks [121] extend this functionality by supporting both the generation of new tokens and the copying of relevant tokens from the input. This approach is particularly beneficial for our code generation task, as there is often significant overlap between tokens in the input and output sequences, such as variable names and method identifiers. These tokens pose challenges for sequence generation tasks due to their infrequent occurrence in small code-description pair datasets. Copy Generator Networks address this issue by providing a mechanism to emphasise tokens irrespective of their frequency in the dataset, effectively copying them from the input. This method enhances the model's ability to handle rare or domain-specific tokens that are crucial for accurate code generation.

Django samples:

```
Desc : description(COPY) is a string "The ’%s’
       function"(COPY) replaced by value of
       receiver(COPY) . __name__ .
Truth: description(COPY) = "The ’%s’ function"(COPY)
       % receiver(COPY) . __name__
Pred : description(COPY) = "The ’%s’ function"(COPY)
       % receiver(COPY)
BLEU : 0.67
```

CoNaLa sample:

```
Desc : split string ` input ` based on occurrences of
       regex pattern ’[ ](?=[A-Z]+\\b)’(COPY)
Truth: re . split ( ’[ ](?=[A-Z]+\\b)’(COPY) , input )
Pred : re . split ( ’[ ](?=[A-Z]+\\b)’(COPY) , input )
BLEU : 1.0
```

Figure 4.5: Multiple predicted samples from the Relevance Transformer on Django and CoNaLa datasets

$$M(w_t|q, c_{0:t-1}) = p_{gen} \cdot T(w_t|q, c_{0:t-1}) + (1 - p_{gen}) \cdot a_t(w_t) \tag{4.4}$$

The implementation of copy generation within our Transformer architecture draws inspiration from the work of [121]. We use the final encoder attention vector to generate a copying vector, which emphasises each input token in proportion to its attention weight $a_t(w_t)$, as shown in Equation 4.4. This copying vector is subsequently interpolated with the original vocabulary distribution $T(w_t|q, c_{0:t-1})$ through a p_{gen} function. Furthermore, our approach incorporates the use of out-of-vocabulary tokens for extremely rare words, as detailed in Section 4.3.2. This mechanism enables a more generalised copying process, allowing for the inclusion of words that may not have been encountered in the training dataset.

4.3 Experimental Setup

This section outlines the datasets, data preprocessing techniques, and evaluation metrics employed in our study.

4.3.1 Datasets

Django [101]

The Django dataset was created by a single engineer who annotated the entire Django source code line by line, resulting in over 18,000 annotated lines. Intended initially for code-to-pseudo-code mapping, this dataset provides detailed descriptions for each line of code.

Hearthstone [86]

This dataset comprises 665 samples derived from the cards of the Hearthstone game. Each sample includes a card's name, description, and key statistics, which collectively form the English description. The corresponding Python source code from the game files serves as the code component. Notably, Hearthstone samples consist of longer sequences, averaging approximately 400 tokens, although many sequences contain similar boilerplate Python code.

CoNaLa [148]

The CoNaLa dataset is sourced from Stack Overflow questions and answers. It contains over 2,000 manually curated, high-quality code-description pairs, which are short answers to programming questions. While the authors provide an additional set of over 600,000 automatically annotated pairs, we determined through evaluation that this automatically annotated set was too noisy for our purposes and opted to use only the 2,000 manually curated pairs.

4.3.2 Data Preprocessing

Our training samples consist of parallel English and code sequences. We process these samples into a common vocabulary set by tokenising based on spaces and specific code identifiers. This tokenisation approach, similar to that used in ReCode [53], preserves strings, variable names, and function identifiers as individual tokens. A unified vocabulary is crucial as it allows for the copying of common tokens shared between input and output sequences. We assign generic positional tokens to out-of-vocabulary tokens shared between sequences, enabling the model to copy potentially unseen relevant tokens to the output based on context. This approach results in a relatively small vocabulary size of under 1,000 tokens while still allowing for the prediction of rare tokens.

4.3.3 Evaluation Metrics

Following the standard practice in the field of code generation [53, 86], we employ BLEU as our primary evaluation metric. We utilise the BLEU implementation from ReCode [53] to assess the quality of our model's output. The scores for each pair are averaged to produce an overall

Description:

```
<sos> get the first object from a queryset
in django model ` Entry ` <eos>
```

Code ground truth:

```
<sos> Entry . objects . filter ( ) [ : 1 ] . get ( ) <eos>
```

Model current decoding sequence:

```
<sos> Entry .
```

Relevant words:

```
[’filter’, ’objects’, ’id’, ’author__id’, ’Book’, ’pk’,
’*’, ’Sample’, ’Entry’, ’name’, "’name’", ’title’,
"’title’", ’exists’, ’-’]
```

Next token prediction:

```
Predicted ’objects’ over ’groupby’
```

Figure 4.6: Generation sample from the Relevance Transformer on the Django dataset. The sample shows a sentence under construction and the token to be produced at the next time step.

BLEU score for the dataset. To ensure the statistical significance of our results, we conduct paired t-tests and apply Bonferroni corrections where appropriate.

4.4 Results

This section presents a comprehensive analysis of our experimental results, structured to empirically validate Hypothesis 1. We first examine the Transformer as a standalone program generator (the macro view), referencing Table 4.1, and then evaluate its augmentation with a retrieval tool (the micro view), referencing Table 4.2.

4.4.1 The Macro View: The Transformer as a Program Generator

To validate Hypothesis 1.1, we first establish the performance of existing methods. Table 4.1 shows that retrieval-based approaches, while simple, provide a competitive baseline. The "Oracle retrieval" scores, in particular, reveal a high performance ceiling (e.g., 74.2 BLEU on Hearthstone), but practical methods like BM25 fall significantly short, indicating that merely finding a similar example is often insufficient. Among generative methods, LSTM-based models represent the prior state-of-the-art for non-AST approaches. Against this backdrop, the baseline Transformer model demonstrates notable efficacy. As shown in Table 4.1, it surpasses all pre-

Retrieval Methods	Django	Hearthstone	CoNaLa
BM25 (fine-tuned baseline)	43.1	59.5	13.2
ReCode sequence similarity	43.4	65.1	11.2
Oracle retrieval similarity	58.1	74.2	38.0
Generative Methods			
Sequence-to-sequence LSTM	58.9	60.4	*10.6*
Latent predictor networks [86]	*77.6*	67.1	—
Retrieve and Edit LSTM [52]	—	*70.0*	—
Transformer [133]	79.2	72.5	17.5

Table 4.1: Performance of baseline retrieval and generative methods. In italic, we show the previous state-of-the-art non-AST methods.

viously reported non-AST results across every dataset: achieving 79.2 BLEU on Django (vs. 77.6), 72.5 on Hearthstone (vs. 70.0), and 17.5 on CoNaLa (vs. 10.6). This strong performance stems from the architecture's self-attention mechanism, which is exceptionally well-suited for code. Unlike LSTMs, which process sequences linearly, the Transformer can model long-range dependencies directly, allowing it to, for example, connect a variable's use late in a snippet back to its initial declaration. This allows it to implicitly learn the syntactic and semantic structure of code without domain-specific components. This result provides direct and compelling evidence for Hypothesis 1.1, establishing the Transformer as a superior and highly expressive method for mapping natural language to code and validating the "program generator" component of our framework.

4.4.2 The Micro View: Augmenting the Generator with a Retrieval Tool

Having established the Transformer's capability, we now address Hypothesis 1.2 by examining the "micro" view, where the generator is augmented by a fixed retrieval tool. The results of these experiments are detailed in Table 4.2. We first note that a simple copy mechanism provides a consistent performance boost over the baseline Transformer, yielding a stronger starting point for our tool use experiments. However, the method of tool integration proves critical. Our "Naïve Retrieval" approach, which makes a hard modification to the input by concatenating the top-retrieved code snippet, severely degrades performance across all datasets, with a dramatic 13.9 BLEU point drop on Hearthstone. This failure occurs because the lengthy retrieved code acts as significant noise, overwhelming the short natural language query. At the time of this research, before the widespread adoption of large-scale pre-training, models like our baseline Transformer were trained only on the specific task data and were thus highly sensitive to such noisy, mixed-signal inputs. In contrast, our proposed Relevance Transformer demonstrates a successful and principled integration. Instead of modifying the input, it provides soft guidance by gently biasing the decoder's output distribution. This approach achieves statistically significant improvements ($p < 0.05$) over the strong Transformer+Copy baseline, reaching new state-

	Django	Hearthstone	CoNaLa
Sequence-to-sequence LSTM	58.9	60.4	*10.6*
Transformer baseline [133]	79.2	72.5	17.5
Transformer + Copy	81.8	74.0	20.8
Transformer + Copy + Naïve Retrieval	80.7	60.1	19.0
Relevance Transformer	**82.3**	**74.5**	**22.3**

Table 4.2: Performance of proposed Transformer-based methods with retrieval integration.

of-the-art scores of 82.3 on Django, 74.5 on Hearthstone, and 22.3 on CoNaLa. As illustrated in Figure 4.6, this mechanism effectively guides the model toward correct token choices—such as predicting objects instead of the plausible but incorrect groupby—by leveraging statistical patterns from relevant code examples. These results confirm Hypothesis 1.2. They not only show that a Transformer's effectiveness can be enhanced by an external retrieval tool but also reveal a crucial insight: for tool use to be effective, the tool's output should assist the model's own reasoning process rather than replacing or overwhelming the original task context.

Figure 4.6 demonstrates the efficacy of our Relevance Transformer in emphasising tokens that are likely to appear in the target sequence. In the provided example, the baseline Transformer predicts the next token in the sequence to be `groupby`. Although this token is contextually relevant, it is not the correct prediction. The pseudo-relevance feedback mechanism addresses this limitation by highlighting common tokens from the top retrieved documents, resulting in the generation of the correct token, `objects`. This example illustrates how the retrieval component enhances the accuracy of the transformer-based model in natural language conditioned program generation.

4.5 Conclusion

This chapter has validated the first component of the thesis statement by demonstrating that language models can be effectively engineered to generate code. We approached this from two perspectives: a macro view focused on identifying a powerful program generator, and a micro view focused on augmenting that generator with an external tool.

First, by addressing the macro view, we provided empirical evidence for Hypothesis 1.1, establishing that the Transformer architecture is a highly effective, syntax-agnostic model for natural language to code translation. Its ability to outperform prior specialised, non-AST methods confirms its capacity to implicitly learn the complex structural patterns inherent in programming languages. This finding provides the thesis with a robust and general-purpose foundation for the program generation tasks explored in later chapters.

Second, by investigating the micro view, we validated Hypothesis 1.2. We introduced the Relevance Transformer, a novel architecture that successfully integrates a fixed retrieval system as a tool to improve code generation. By using pseudo-relevance feedback as a form of "soft guidance," our model intelligently incorporates information from external code snippets, leading to state-of-the-art performance. The failure of a naïve concatenation approach underscored a critical insight: the method of tool integration is as important as the tool itself.

By successfully demonstrating that a language model can both generate code and leverage a simple tool to improve that generation, this chapter lays the essential groundwork for the more complex, dynamic tool use paradigms explored in the rest of this thesis. The ability to reliably translate intent into executable programs is the prerequisite for enabling language models to externalise reasoning and interact with structured environments, a capability we will now extend to the challenging domains of tabular question answering and conversational AI.

4.6 Broader Context and Subsequent Developments

The central bet of this chapter, that a general-purpose, syntax-agnostic architecture is sufficient for natural language to code translation, was validated decisively in the years that followed. OpenAI Codex [20], fine-tuned on a large corpus of public GitHub code, demonstrated that the same Transformer family scales to generate complex, multi-file programs with far greater fluency than any domain-specific approach achievable in 2019. GitHub Copilot subsequently deployed this capability to millions of developers, making AI-assisted code generation a standard part of the software engineering workflow. The thesis was early principled evidence for the right architecture at a time when AST-based, domain-specific models were the assumed requirement.

Chapter 5

Weak Supervision for Language to Code Translation

This work was carried out in 2020–2021, with the conversational components published at TREC 2020 [43]. The previous chapter established that the Transformer can semantically translate natural language to code, but two questions remained open: whether performance could be improved without additional labelled pairs, and whether the generated programs were syntactically valid. At that time, large-scale pre-training with masked language modelling objectives had proven effective for instilling generalised language skills in downstream tasks [16, 32, 89], and BART [81] had extended this to generative sequence-to-sequence settings. CodeBERT [36] had demonstrated that code pre-training was feasible but focused on understanding rather than generation.

This chapter examines how sources of weak supervision can improve natural language to code translation. It studies the choice of fine-tuning objective when adapting pre-trained models, the generation of synthetic parallel data from large language models, and the effect of source material on downstream performance. It shows that infilling objectives are superior to direct translation fine-tuning, and extends the evaluation of Chapter 4 by measuring syntactic correctness alongside semantic similarity.

Within the context of the framework described in Chapter 2 we are improving the performance of the program generator as seen in Figure 5.1.

5.1 Introduction

Pre-training consists of training the weights of a neural network to model a distribution of data prior to task-specific fine-tuning. In the case of language models, this pre-training is typically achieved through a combination of causal language modelling and masked language modelling objectives. Although all objectives follow the same principle of reconstructing one or more masked tokens from the surrounding context, the choice of objective can have a significant

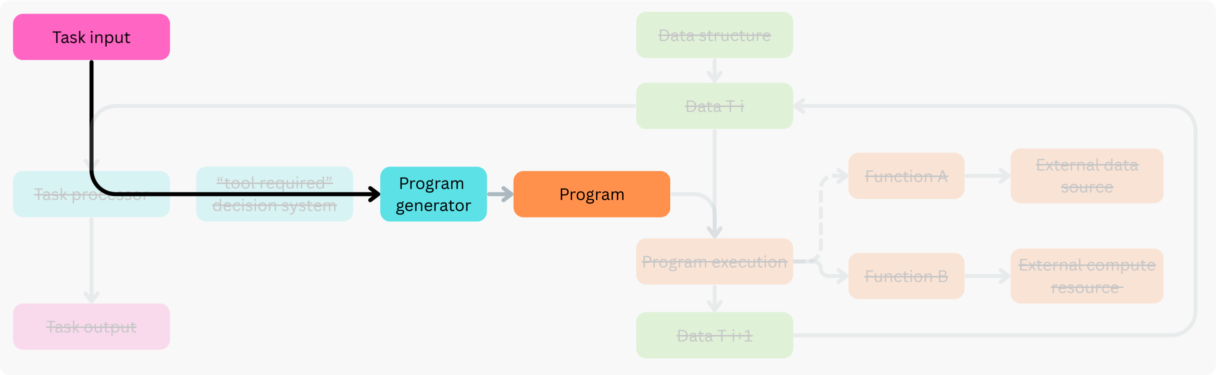


Figure 5.1: The tool use diagram highlighting the components that are studied in this chapter. **No data structures are used since there is no program execution. The task input is a natural language statement describing the desired behaviour of a Python program. The program generator is the sole component studied in this chapter. Its task is to translate the natural language statement into a valid Python program. Although the produced program is not executed, it is still measured for syntactic and semantic correctness.** All other components of the diagram are modelled in later chapters.

impact on the downstream task performance.

English and code can co-occur in large-scale corpora (such as Stack Overflow) during pre-training. As a result, pre-trained language models can have latent alignment between English and code. In this chapter, I explore how to effectively use this latent knowledge in combination with the pre-training objective for the task of natural language to code generation. As a result, this chapter directly addresses the hypothesis: *NL to code translation can be improved through sources of weak supervision.*

In order to address this overarching question, this chapter is structured as follows:

1. **Infilling objectives improve task-specific performance:** As preliminary work to prove the concept of aligning pre-training objectives to the downstream task objective, we study the task of conversational query rewriting. This allows us to isolate the study to a single modality (English) while testing translation vs infilling for BART [81] models. This sets the stage to answer our hypotheses.

2. **Infilling objectives for NL to code translation:** To address Hypothesis 2.1 *Infilling training objectives are superior to translation objectives for NL to code translation* I use a corruption pre-training objective which involves reiterating known context before predicting, and confirm the trend shown prior holds for NL to code translation.

3. **Joint code and English pre-training:** To address Hypothesis 2.2 *Infilling pre-training that jointly considers aligned natural language (NL) and code modalities is more effective than independent modality pre-training for NL to code generation*, we perform ablation studies using the CodeSearchNet corpus and compare the downstream model effectiveness.

4. **Generating weak supervision data**: Finally, we address Hypothesis 2.3 whereby *language models can be used to generate weak supervision data to improve the effectiveness of NL to code translation systems* by leveraging synthetically generated data as a means of instilling knowledge gained during pre-training for NL to code translation. This section also explores the impact of using different source code corpora for generating these synthetic NL descriptions.

To strengthen the parallel between the two tasks covered in this chapter, we explain below how to unify the inputs and outputs. This unification is possible since we model both tasks as a sequence-to-sequence problem, allowing us to transfer improvements from one task to the other:

- **Input**: The input to both tasks is English text. For conversational query rewriting, it is an unresolved question like *"Why doesn't it spoil?"* and in code translation, it is an English description of a program like *"sort 2d array ‘matrix‘ by row with index 1"*.
- **Output**: The output is also represented as text. For conversational query rewriting, it is the resolved question like *"Why doesn't honey spoil?"* and in code translation it is the equivalent of a Python program like `sorted(matrix, key=itemgetter(1))`.

5.2 Infilling Objectives Improve Task-specific Performance: Conversational Query Re-writing

It may seem counterintuitive to discuss conversational query rewriting in a chapter about code translation. However, to validate our proposed reiteration objective for the BART pre-trained model, we leverage an initial study in the task of conversational rewriting that seeded the idea of reiteration. Similar to natural language to code translation, conversational rewriting requires the output sequence to be heavily influenced by the input sequence. We use CAsT (Conversational Assistance Track) at TREC to empirically test that aligning the pre-training objective of a BART model with its fine-tuning objective can be more effective than a traditional translation objective for this model.

5.2.1 Task Description

Similar to traditional document retrieval, CAsT is framed as an information-seeking task where the objective is to return a relevant document $d*$ from a large corpus D given a query q. However, the conversational nature implies that at certain turns, not all the required information to satisfy the user's query will be contained in q. We formalise the sequence of raw turns as:

$$Q_n = \{q_1, q_2, \dots, q_n\}$$

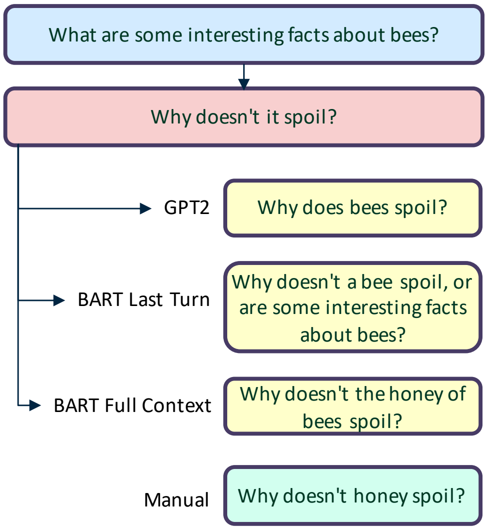


Figure 5.2: A comparison of the three query rewriters on a CAsT 2020 topic.

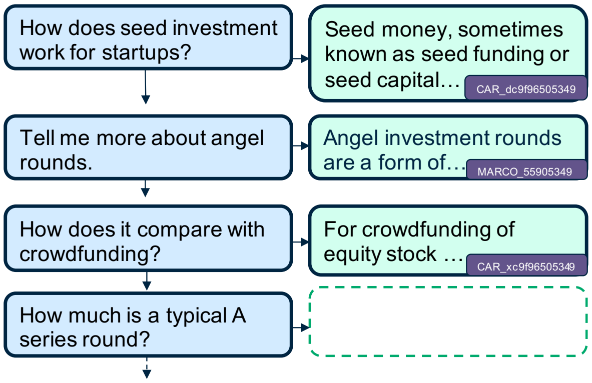


Figure 5.3: A conversation sample from CAsT 2020.

Similar to human-human interaction, not all information is provided during a single utterance and is spread through the preceding turns of the conversation. This, in turn, leads to an information dependency for any q_i on any number of preceding turns. We denote $q*$ as a fully resolved query containing all information necessary to obtain a complete set of relevant documents, $d*$, without the need for additional context. We define an aggregation function f such that all information contained in the sequence Q_n results in $q*$. We note that while there is much information in Q_n, not all of which is relevant at turn n, the aggregation function f specifically condenses relevant information for the last turn q_n.

$$q* = f(Q_n) \tag{5.1}$$

This spread of information across queries leads to a breakdown of traditional search systems reliant on a concise query $q*$.

In comparison to CAsT 2019, the 2020 version of the task increases the information dependency on the corpus D by segmenting the information needed to obtain $q*$ in the results of previous turns. Thus, the previous results denoted as $d*_{1,\dots,n-1}$ are for part of the inputs for f

$$q* = f(Q_n, d*_{1,\dots,n-1}) \tag{5.2}$$

This increases the fidelity of the task with respect to real-world conversations, and increases the difficulty significantly since a theoretical marginalisation over D is required, yet D is often many orders of magnitude larger than Q_n.

Due to the low frequency of situations where $q*$ is dependent on $d*_{1,\dots,n-1}$, we opt to model f as we describe in Equation 5.1 by distilling all relevant information from Q_n to approximate q_*.

Challenges in CAsT are categorised into two types: manual runs and automated runs. Manual runs evaluate a system's performance and retrieve $d*$ from D when $q*$ is given, thus testing raw retrieval capacity. Automatic runs simulate the full conversational scenario by providing a sequence of unresolved queries Q_n. To achieve the best performance for the task, we optimise every section of the conversational retrieval pipeline. Our final system consists of multiple stages of retrieval and re-ranking to complement our initial rewriting strategy. In this work, we focus solely on the Automatic challenge.

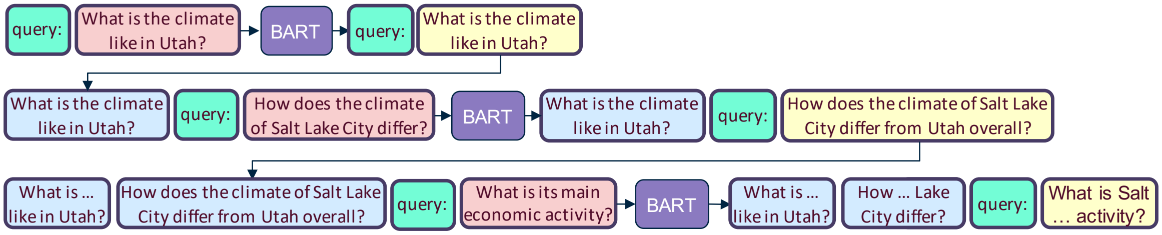


Figure 5.4: Example of full contextual rewriting with BART

- **GPT2_duo_norm**: GPT-2 rewriter with a BM25, mono & duoBERT ranking pipeline. Pair-wise scores are normalised in the original list.
- **grill_BART_LT_bm_duo_norm**: An automatic run with a BART-LT rewriter generating only the resolved query, BM25, mono & duoBERT ranking pipeline.
- **grill_BART_FC_bm_duo_norm**: An automatic run with a BART-FC rewriter, BM25, mono & duoBERT ranking pipeline.

5.2.2 Query Re-writing

In the following section, we outline our proposed method f' to aggregate information from Q_n to approximate $q*$.

Since we hope to obtain a new query q' from Q_n, an ordered set of unresolved queries, we opt for a sequence-to-sequence model as a rewriter. We use the Transformer [133] architecture to benefit from increased parallelism during training. CAsT is a challenging environment for these types of architectures, given the limited dataset size. Pre-trained architectures [32] have been shown to remedy this to an extent by leveraging general text features; as such, we initialise our rewriter with pre-trained weights from BART [81] trained on text from Wikipedia and fine-tuned on a summarisation task for CNN & Daily Mail. While weights from Wiki BART could already encode meaningful features, the training is framed as a span infilling task. To increase the information extraction and synthesis capabilities of our model, we use the summarisation weights. We keep the same vocabulary to ensure feature transfer and use a conditional sequence modelling objective with cross entropy loss as the gradient signal for 500 context and query samples from CAsT 2019 & 2020.

In order to indicate to our model the origin of the information requirement as the last query in the sequence q_n, we highlight q_n by inserting a constant token sequence [query:] prior to the unresolved query during training and inference. This allows us to frame the input to f' as two types of inputs: Q_{n-1} as context and q_n as the unresolved last query. We model the rewriting process as follows:

$$q_* \approx q' = f'(Q_{n-1}, q_n) \tag{5.3}$$

To remain consistent with the original pre-training objective, we keep the reconstruction objective by tasking the model to reproduce the original sequence of context queries as well as the rewritten final query. We use the name BART-FC (BART Full Context) to refer to this form of inference.

$$f'(Q_{n-1}, q_n) = cut(BART(Q_{n-1}, q_n)) \tag{5.4}$$

This ensures we benefit from the language modelling objectives of BART rather than imposing a new sequence structure where the only output is q_n. We then truncate the final output with a function *cut* to only keep q_n as seen in Equation 5.4. To evaluate the effectiveness of BART-FC against a more traditional sequence-to-sequence formulation we create BART-LT (BART Last Turn) where the only generated output is the target resolved query.

Relevant information for resolving q_n might be implicitly present multiple times in Q_n as recurring entities and themes propagate through the conversation. However, an explicit mention might only appear once towards the beginning of the conversation, similar to a human-human conversation. This poses an additional burden on our model to follow the chain of dependencies implicitly. While bidirectional information flow in the Transformer encoder allows for such information to be obtained, we opt to highlight common threads in the conversation by feeding resolved queries $Q*_{n-1}$ as context to resolve a unique ambiguous query q_n. At inference time, this results in auto-regressive synthesis at a sentence level according to Equation 5.5.

$$q'_n = f'(q'_1, q'_2, ..., q'_{n-1}, q_n) \tag{5.5}$$

5.2.3 Second Pass Re-ranking

The quality of a rewritten query is exposed through the documents that are retrieved from it. Given a pool of retrieved documents with respect to q, we use multiple re-rankers to re-order documents according to relevance. In the following section, we outline our approach.

Mono BERT

In order to capture semantic relationships between a query and relevant documents, we perform a point-wise re-ranking using monoBERT [97]. MonoBERT performs a bi-encoding of the query and document and uses a custom classification head on the [CLS] token to score s the relevance of the document to the query. Given the similarity of the training and target domains on MS Marco, we do not perform any further fine-tuning to the model for CAsT. The new point-wise scores are then used to re-order the documents.

Duo BERT

DuoBERT from Nogueira [99] is a pair-wise comparison model that evaluates the relevance of two documents d_a & d_b against a query q. All three sequences form part of the input to BERT with a specific head on the [CLS] token indicating a weighting to either document. This process allows a grounded comparison rather than a floating score given by monoBERT.

Since duoBERT is a pairwise model, scores are only relative and thus scores for a document are obtained by summing its given score when compared to all other documents. This is an expensive process with $O(l^2)$ time complexity with respect to the pool length l and as such is only suited as the last stage of a re-ranking pipeline. In our experiments, we find a good trade-off between pool length and time by taking the top 10 documents from the monoBERT scores.

These re-ranking stages ensure that the highest quality documents are pushed to the top of the ranking.

Model	**NDCG@3**	**NDCG@5**	**NDCG**	**MAP**	**R@500**	**R**
GPT2_duo_norm	0.313	0.300	0.289	0.164	0.333	0.333
BART_LT	0.336	0.324	0.389	0.219	0.504	0.516
BART_FC	0.398	0.379	0.417	0.239	0.515	0.526

Table 5.1: Performance comparison for CAsT 2020 automatic runs. Metrics without depth indication are taken at 1k. R indicates recall.

Table 5.1 demonstrates the importance of query rewriting for improved performance. GPT-2, a powerful language model developed by OpenAI, serves as a baseline for comparison. Despite its impressive capabilities in various natural language processing tasks, GPT-2 shows limitations in this specific query rewriting scenario.

BART-LT outperforms GPT-2 when given the same training objective of generating solely the resolved query. BART-FC, which remains more closely aligned with its original pre-training objective during fine-tuning and inference, yields the best results. This model shows both qualitative and empirical improvements when using the same re-ranking pipeline. As observed in Figure 5.3, BART-FC demonstrates deep semantic pattern recognition in resolving queries. It leverages grammatical properties and open-domain knowledge, such as understanding that "spoil" doesn't apply to "bees" and refers to the singular noun "honey". This capability is particularly impressive given that "honey" was not explicitly mentioned in the preceding context.

Given that the only difference between BART-FC and BART-LT is the reconstruction objective, the superior performance of BART-FC can be attributed to its ability to utilise the knowledge captured in pre-trained weights for effective knowledge transfer. This is a direct result of maintaining the original reconstruction objective rather than fine-tuning using the traditional translation objective, which is especially beneficial when working with limited dataset sizes.

These findings have important implications for code generation tasks. By leveraging pre-trained language models like BART and applying similar principles of maintaining pre-training objectives during fine-tuning, we can potentially improve the quality and accuracy of code generated from natural language descriptions. This approach could lead to more context-aware and semantically accurate code generation.

5.3 Infilling Objectives for NL to Code Translation

In this section, we apply our findings from conversational query rewriting to the task of natural language to code generation. Given the structured nature of code, research in this area has invested significant effort in utilising program-specific knowledge to optimise these systems. Hand-crafted architectural choices addressing grammatical and syntactic constraints are widely adopted in methods for code generation. While the domain knowledge yields improvements over generic models, their complexity and language specificity require significant effort to modify and extend.

5.3.1 Task Definition

We define the task of generating code from a natural language (NL) description as the production of a sequence that satisfies the described functional intent and is compliant with a programming language's (PL) syntactic and grammatical rules.

Given an NL description $d = [d_1, d_2, ..., d_n]$ of length n and a target code sequence $c = [c_1, c_2, ..., c_m]$ of length m, we define f with trainable parameters θ as the mapping function between them. For each instance of d, there is a single correct target sequence, which we denote as $c*$. When training our systems, we formulate the optimisation task as follows:

$$c* \simeq c = f(d, \theta)$$

We consider $f(\theta)$ optimal when a similarity metric s is maximised with respect to other parameter configurations of θ for a corpus of parallel samples C and D.

$$s = similarity(f(D, \theta), C*)$$

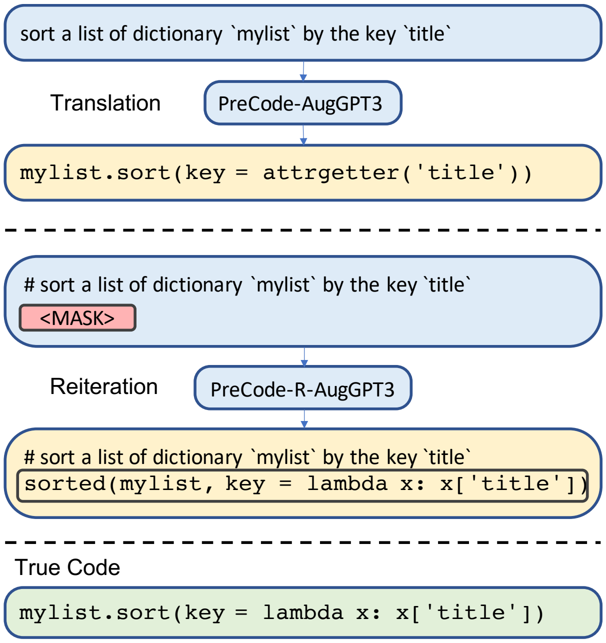


Figure 5.5: An example of the translation and reiteration objectives applied to the task of code generation.

5.3.2 Methods

In order to focus the BART model on the task of code generation conditioned on a natural language description, we fine-tune it with two differing objectives: translation and reiteration. Both objectives use a sequence cross-entropy loss to maximise the conditional probability of a sequence of tokens.

Translation

Our translation fine-tuning objective is akin to many works in code generation literature [86, 101, 111, 148] in that it models a sequence transduction from a language in one domain to another. This is effectively a transformation of the original input d with the model acting as intermediary, where the amount of information remains constant on both sides of the process. We translate from the input description d to the target sequence c.

Reiteration

In order to benefit maximally from the knowledge acquired during pre-training, we introduce a reiteration objective for conditional sequence generation. This fine-tuning objective is specifically designed to align with the distribution of data seen during pre-training by framing the code generation task as text infilling.

We define the function *reiterate* as taking a template T and the description d to simulate a corrupted span of code. Reiteration aligns the fine-tuning objective by framing the description

Method	BLEU	Parses
BART + Translation	14.53	44.8%
BART + Reiteration	18.61	45.1%

Table 5.2: Comparison between translation and reiteration fine-tuning objectives for BART on CoNaLa.

as a comment in the code defined in the template. The location of the code to be generated is marked with the <MASK> token following a line break, similar to how a single line could be commented in real code. When *reiterate* is given the correct code sequence $c*$, we substitute the <MASK> sequence in the template T.

```
T = "# [d]\n<MASK>"
```

$$i' = reiterate(T, d)$$

$$i* = reiterate(T, d, c*)$$

Reiteration then optimises f as described in Equation 5.7. When i is produced, we then use *clean* to extract the code from the generated sequence, given the template. This process is depicted graphically in Figure 5.5 and formally in Equation 5.6

$$c* \simeq c = clean(T, f(reiterate(T, d))) \tag{5.6}$$

5.3.3 Experimental Setup

We use BART [81] as our general sequence-to-sequence architecture when applying reiteration and translation for code generation. During fine-tuning, we use a learning rate of $5 \cdot 10^{-5}$ and a warm-up of 500 steps with exponential decay. We do early stopping according to the validation sets provided and report on the test set.

We use the CoNaLa test set, described in detail in Section 4.3.1, for evaluation. We use BLEU [103] between the predicted and ground truth code to measure semantic similarity and separately check whether the code parses according to the Python syntax.

5.3.4 Results

As seen in Table 5.2, we observe that reiteration has a significant impact on semantic similarity for code. This is surprising since BART is not explicitly trained on code. In combination with our prior results in conversational rewriting, this shows that aligning pre-training and fine-tuning objectives is effective for model specialisation.

When investigating syntactic correctness, we observe minor improvements of 0.3% absolute. This is likely to be noise during evaluation and as such confirms that much of the syntactic knowledge for code generation is obtained during the fine-tuning stage, irrespective of the optimisation objective.

These results directly validate Hypothesis 2.1 *Infilling training objectives are superior to translation objectives for NL to code translation*.

5.4 Joint Code and English Pre-training with Infilling

At this point, it is clear that the pre-training data and objective have a significant effect on the downstream performance of a model. In this section, we continue the pre-training stage of the BART model with data in the modalities of interest: English and code.

In particular, we study the effect of jointly aligning these modalities through the use of specialised corpora to confirm effective ways for language models to generate code from English descriptions. Figure 5.6 shows an input-output instance used during continued pre-training.

Data

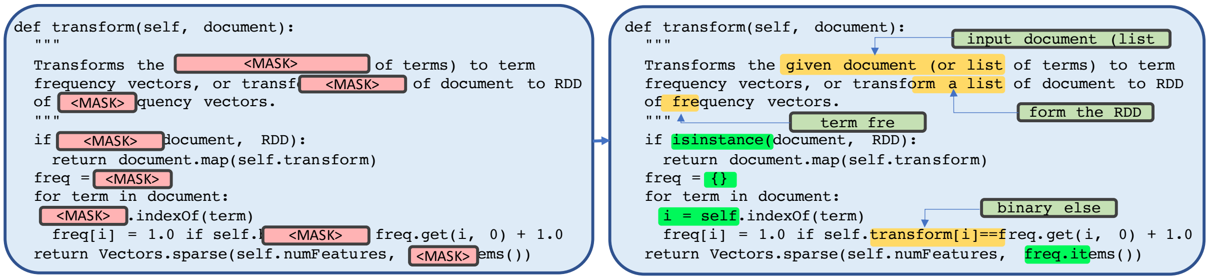
We use the CodeSearchNet corpus [61] as our source of pre-training data. This corpus consists of 184916 Python functions with docstrings sourced from GitHub. When pre-training our models, we make three variants of this data to evaluate the utility of different data:

- **Only code**: We keep only code segments of the Python CodeSearchNet corpus by removing the docstrings. In doing so, we expose the model to solely the grammatical and syntactic rules of Python while reducing exposure to connecting English words.
- **Only English**: We keep only docstrings for each function in isolation. We opt for our own English language pre-training since other checkpoints, such as BART-base, could suffer from data distribution shift with respect to the English used for code, different tokenisation strategies, and unequal training resources.
- **Full function**: The full function and docstring are used together as a unique sample. By using it for both sequence understanding and generation, we condition the generated spans on the surrounding code and English.

5.4.1 Method

In order to distinguish the different pre-training stages, we define the following conventions:

- Methods containing **solely BART** refer to pre-trained models deriving from the English-only corpus from the original work in [81].



```
def transform(self, document):
  """
  Transforms the <MASK> of terms) to term
  frequency vectors, or transf<MASK> of document to RDD
  of <MASK>quency vectors.
  """
  if <MASK>document, RDD):
    return document.map(self.transform)
  freq = <MASK>
  for term in document:
   <MASK>.indexOf(term)
    freq[i] = 1.0 if self.<MASK> freq.get(i, 0) + 1.0
  return Vectors.sparse(self.numFeatures, <MASK>ems())
```

```
def transform(self, document):
  """
  Transforms the given document (or list of terms) to term
  frequency vectors, or transform a list of document to RDD
  of frequency vectors.
  """
  if isinstance(document, RDD):
    return document.map(self.transform)
  freq = {}
  for term in document:
   i = self.indexOf(term)
    freq[i] = 1.0 if self.transform[i]==freq.get(i, 0) + 1.0
  return Vectors.sparse(self.numFeatures, freq.items())
```

Figure 5.6: An example showing the variable span infilling pre-training objective used with the full function Python data from CodeSearchNet. Green (exact match) and yellow spans indicate generated sequences by PreCode. We note that all <MASK> sequences occur simultaneously. Bordered sequences indicate the ground truths for a span.

- Methods containing **PreCode** are checkpoints that have been further trained on the CodeSearchNet collection with a corruption objective.

Formally, we define the PreCode transformation as the corruption of a code sequence i through a function *corrupt* to hide information with a specific <MASK> token noted as i'. Spans of arbitrary length within a minimum and maximum are removed from the original sequence and substituted until reaching a corruption rate of 15%. Our span selection process occurs at a character level, resulting in a sequence of tokens the model might not have been exposed to had whole words been maintained.

$$i' = corrupt(i)$$

The model is then tasked to reconstruct the original sequence auto-regressively. An example on the CodeSearchNet dataset is shown in Figure 5.6.

$$i* \simeq i = f(i', \theta) \tag{5.7}$$

Importantly, the model during pre-training is tasked to reiterate the input sequence while filling in the blanks specified by the masking tokens rather than solely producing the missing target sequences. This reiteration is an important aspect that we benefit from during fine-tuning.

PreCode models are exposed to the text infilling task in the domain of code since every sample is a complete function. When predicting a masked span, it can rely on structurally consistent elements such as docstrings or line comments with surrounding markers. In addition, the masking tokens signal to the model the location of a missing token sequence. These are elements that are omitted when using a translation objective since neither code structure nor masked token signalling is preserved.

Model	BLEU	Parses
BART	14.53	44.8%
BART + PreCode	31.32	87.0%
BART + Reiteration	18.61	45.1%
BART + PreCode + Reiteration	**32.01**	**92.6%**

Table 5.3: Comparison of adding PreCode (full function pre-training) to BART with different fine-tuning objectives on CoNaLa.

Model	BLEU	Parses
BART	14.53	44.8%
BART + PreCode (only Eng)	19.16	45.8%
BART + PreCode (only code)	24.37	81.1%
BART + PreCode (full)	**31.32**	**87.0%**

Table 5.4: Comparison of different PreCode pre-training data modalities on CoNaLa. All methods use the translation fine-tuning objective.

5.4.2 Results

In Table 5.3, we can observe the impact PreCode has on downstream models. We see that regardless of the downstream objective, PreCode yields large improvements for both semantic similarity to ground truth code and syntactic correctness. In the case of reiteration, we observe the largest increase in syntactic correctness with a 47.4% absolute improvement, further strengthening the case for aligning pre-training and fine-tuning objectives for NL to code translation.

In order to better understand what enables this improved performance, we perform an ablation study on the different modalities within the CodeSearchNet dataset. We use our defined splits (only English, only code, and full) as well as a control where we don't apply any further pre-training on top of BART. All of the methods in Table 5.4 are fine-tuned using the translation objective. We observe various phenomena:

1. Pre-training on docstrings in the English-only case can improve semantic similarity but has little effect on code syntactic correctness.

2. Pre-training on the solely function body has the largest relative effect on syntactic correctness by increasing it from 45.8% to 81.1%. This also has a marked positive effect on semantic similarity, with a 5.21 BLEU score absolute increase.

3. Pre-training on functions and docstrings jointly has the biggest relative impact in semantic similarity, with a 6.95 BLEU score increase over just using the code alone. Furthermore, this addition of natural language does no harm to syntactic correctness and even improves it by a further 6%.

Overall, we show that sources of weak supervision, such as functions with associated docstrings, can be very effective in improving syntactic correctness and semantic similarity for the task of code generation from NL descriptions. This allows us to confidently validate Hypothesis 2.2 *Infilling pre-training that jointly considers aligned natural language (NL) and code modalities is more effective than independent modality pre-training for NL to code generation.*

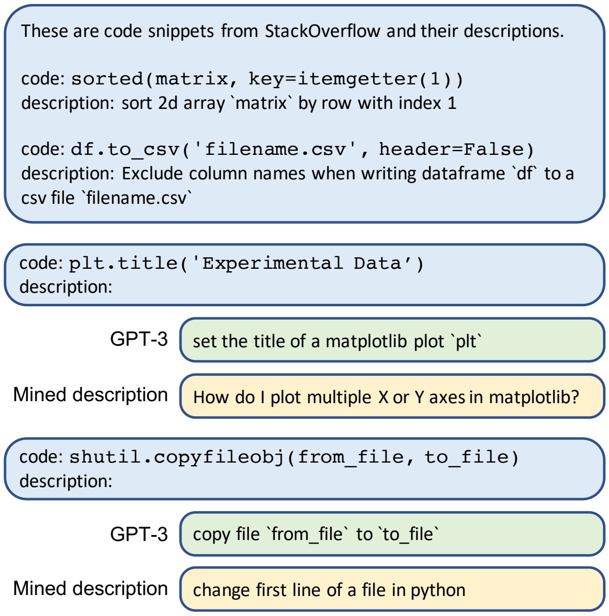
5.5 Generating Weak Supervision Data for NL to code Translation

As we've seen in the previous sections, pre-training is a powerful method to improve task-specific performance in language models. One of the most notable and general examples of this paradigm is GPT-3 [16], a large language model trained on a variety of English data from the web. In this section, we study the effect of using GPT-3 as a data augmentation method to leverage its few-shot learning capabilities, extending this to various source code corpora beyond the initially explored CoNaLa Mined set.

GPT-3 is trained in a unidirectional manner to provide token probabilities given a previous sequence. Through extensive qualitative analysis, it has been shown that GPT-3 is able to effectively take prior context into account and learn from a few samples of data. The prior context is framed as a prompt, both explaining the task with an English description and providing several examples of the task to follow. The size of the prompt is limited by the context window of GPT-3 and the length of the desired output. We use the largest model variant, Darwin, with 2048 tokens of context available for the prompt and the generated sequence.

In order to obtain samples to synthetically annotate, we use the various subsets of CoNaLa. The CoNaLa corpus [148] is a parallel collection of Python 3 snippets with associated English descriptions that was obtained by automatically mining 500k Stack Overflow posts with a classification model judging the quality of the match between the title of the thread used as the description and the top-voted code segment. The classification algorithm was trained using a manually curated collection, resulting in a high-quality parallel corpus. After splitting for validation and testing, this results in a collection of 500 (test), 238 (validation), 2140 (train), and 500k (mined) samples.

The 500k mined samples can introduce excessive noise in the training process and are often omitted. When analysing the samples individually, we observe a frequent mismatch between the description and the associated code, and use GPT-3 to recreate the English description given the code. We formulate the task as generating an English description given a code snippet. We depict an example of this in Figure 5.7.



These are code snippets from StackOverflow and their descriptions.

code: `sorted(matrix, key=itemgetter(1))`
description: sort 2d array `matrix` by row with index 1

code: `df.to_csv('filename.csv', header=False)`
description: Exclude column names when writing dataframe `df` to a csv file `filename.csv`

code: `plt.title('Experimental Data’)`
description:

GPT-3: set the title of a matplotlib plot `plt`

Mined description: How do I plot multiple X or Y axes in matplotlib?

code: `shutil.copyfileobj(from_file, to_file)`
description:

GPT-3: copy file `from_file` to `to_file`

Mined description: change first line of a file in python

Figure 5.7: GPT-3 few-shot prompt used to generate descriptions for a single Python snippet. The full prompt contains 20 code-description pairs from the CoNaLa training set.

5.5.1 Experimental setup

We create the prompt by describing the task to be accomplished and 20 examples of code with associated descriptions from the CoNaLa training set. This primes GPT-3 with several instances of correct descriptions to generate. It is important to note that the provided prompt is not exhaustive and that other primers could also lead to similar results when designed for GPT-3.

During the creation of this augmented set, we note a slight decrease in quality beyond 50k samples. This is due in part to the ordering of the mined samples from the CoNaLa set, which are ordered by relevance between the code and the mined description. We determine that the highest quality code samples are located at the top of the list and, as such, conduct our experiments with the top 50k samples annotated by GPT-3 denoted as AugGPT3. This AugGPT3 specifically refers to synthetic NL generated for the CoNaLa Mined code.

We use GPT-3 [16] as a baseline for comparison as a large-scale general sequence-to-sequence language model. Given the nature of the few-shot prompt, reiteration is implicit at inference time because the desired token sequence is a direct continuation of prior context.

Similar to our data augmentation procedure, we leverage its few-shot capabilities to generate code given a description. We reverse the direction of the prompt, stating a description of the task and several examples. We randomly sample 20 to 30 examples from the CoNaLa training set to saturate up to 80% of the 2048 token limit for the prompt. When evaluating on a test set, we append the input description with the same pattern as prior samples and prime the last token before generation.

We use the Open-API provided by OpenAI to conduct our experiments with 50 parallel requests to the 175B Darwin variant of GPT-3. We use a temperature and top-p sampling of 0 to remain deterministic and maintain reproducibility.

For fine-tuning, we use the same training setup for BART as in the prior sections, specifically using the translation objective for experiments detailed in this subsection unless otherwise noted.

TranX

To further validate that language models can synthetically generate data that improves the performance of NL to code models, we apply the same AugGPT3 data (from CoNaLa Mined) to fine-tune a TranX model. At the time of writing, TranX [150] is the state-of-the-art algorithm for code generation on CoNaLa. It is an RNN-based architecture specifically designed for code generation from natural language, which boasts many features tailored to the generation of structured programs, such as:

- Parent hidden state transfer to lower nodes according to a programming language's AST. By feeding parent information to child hidden states, gradient information regarding node types can flow more efficiently during training to elements in the sequence.
- Constrained decoding: avoiding incorrect tree generation at inference time by taking into account syntactic and grammatical rules.
- Transparent token copying from input to output. This substitutes possible tokens to be copied by generic sequential tokens, abstracting the contents to avoid rare tokens during inference time.

These architectural choices reflect domain-specific attributes for the task of code generation. By including these, knowledge from the designers familiar with the task is instilled implicitly in the model.

We include the base variant of TranX from their publication in our comparisons, as well as our reproduction of their results, which is trained on additional external knowledge. This allows us to compute the syntactic correctness. When training on the GPT-3 annotated samples, we are consistent with the training procedures provided by the authors for large-scale pre-training [146], and use the same Corpus BLEU [103] to evaluate sequence similarity. We also state the parsing ratio of produced samples by using the AST module provided in Python 3. We note AST parsing does not result in functional code, as run-time errors are not checked.

5.5.2 Results

Table 5.5 shows how the addition of our synthetically generated English-code pairs (AugGPT3, derived from CoNaLa Mined) affects our previously described methods. We observe that semantic similarity and syntactic correctness consistently improve whether we use reiteration or translation. Surprisingly, the biggest gains are in non-reiteration methods, which catch up to

Model	BLEU	Parses
GPT-3	32.81	93.0%
BART + PreCode	31.32	87.0%
BART + PreCode + Reiteration	32.01	92.6%
BART + PreCode + **AugGPT3**	33.32	94.0%
BART + PreCode + **AugGPT3** + Reiteration	**33.92**	**94.2%**

Table 5.5: Performance of baseline systems and models fine-tuned with synthetic data (AugGPT3 from CoNaLa Mined) on the CoNaLa test set.

Model	BLEU	Parses
TranX	27.20	—
TranX + Ext. knowledge	30.69	98.8%
TranX + AugGPT3	**32.21**	**99.6%**

Table 5.6: Performance of TranX models on CoNaLa, including with synthetic data (AugGPT3 from CoNaLa Mined).

match the performance of their reiteration counterparts. This shows that in the face of 50k synthetically generated samples, the alignment of pre-training and fine-tuning objectives is less important. These findings emphasise the effectiveness of language models to generate synthetic data for the purposes of NL to code translation.

The results for TranX (Table 5.6) show a similar story where the addition of AugGPT3 data improves on all fronts. This is surprising as TranX is a specialised code generation model with abstract syntax tree decoding, yet the syntactic correctness improves nonetheless. We hypothesise that it may be due to a broad exposure to known correct code since AugGPT3 consists of synthetically generated English from mined code.

We demonstrate, using Table 5.7 and Figure 5.6, the domain-specific skills that are learned as a direct result of joint code & English pre-training and fine-tuning using synthetic data.

Variable Copying

The ability of a model to identify and copy sections of input is instrumental in the domain of code generation. Method names, variables, and literals (strings, floats, and integers) are often elicited as affected objects in the target code snippet. Moreover, copyable sections are variable in length with phenomena such as spaces, commas, and quotes interrupting a contiguous sequence of characters.

PreCode trained models have no explicit copying mechanism by design and thus must learn this ability through data interaction. In table 5.7, rows 2,3,4 & 6 show perfect copying abilities with GPT-3 missing a single example in row 4. These are complex multi-token sequences with varying levels of randomness. While `Output.txt` could be common in large code corpora, sequences like `4a4b4c` are rare during generation and require internal information routing to

NL description	**get the date 7 days before the current date**
True code	(datetime.datetime.now()-datetime.timedelta(days=7)).days()
Ours	now = datetime.datetime.now() - datetime.timedelta(days=7)
TranX	datetime.datetime.now()
GPT-3	datetime.date.today() - datetime.timedelta(days=7)
NL description	**open a file ’Output.txt’ in append mode**
True code	file = open(’Output.txt’, ’a’)
Ours	f = open(’Output.txt’, ’a’)
TranX	f = open(’Output.txt’, ’r’)
GPT-3	with open(’Output.txt’, ’a’) as f:
NL description	**set pythonpath in python script**
True code	sys.path.append(’/path/to/wherever’)
Ours	sys.path.append(’/path/to/python’)
TranX	os.chdir(os.path.dirname(__file__))
GPT-3	sys.path.append(’/path/to/python/module’)
NL description	**convert ’123,456.908’ into a floating number**
True code	float(’123,456.908’.replace(’,’, ”))
Ours	print(float(’123,456.908’))
TranX	print(’, ’.join(map(str, 123,456.908)))
GPT-3	float(str(123456.908))
NL description	**concat two matrices ’A’ and ’B’ in numpy**
True code	np.vstack((A, B))
Ours	np.hstack((A, B))
TranX	np.importlib(A <B, axis=0).B()
GPT-3	A.concatenate((B, B))
NL description	**decode a hex string ’4a4b4c’ to UTF-8.**
True code	bytes.fromhex(’4a4b4c’).decode(’utf-8’)
Ours	"""4a4b4c""".decode(’UTF-8’)
TranX	print(4a4b4c.decode(’4a4b4c’))
GPT-3	’4a4b4c’.decode(’utf-8’)

Table 5.7: Ours refers to PreCode + AugGPT3 + Reiteration. Randomly selected code generation samples from the CoNaLa test set.

pass input token representations transparently to the decoder for blind generation. In addition, knowing when to stop copying is demonstrated by paying attention to cues such as quotes. This is a skill handcrafted into a task-specific model like TranX.

Syntax and Grammar Awareness

Figure 5.6 depicts a direct output of a resulting PreCode training from masked spans. Along with deep semantic knowledge from filling in program sections accurately, we observe syntactic proficiency in PreCode models through

- Parenthesis matching on multiple occasions

- Inclusion of common Pythonic patterns like `self`
- Awareness of standard Python functions like `isinstance`, which do not appear in the original code
- Variable use consistent with other parts in the function, such as `transform`, `i`, and `freq`

Overall, we see encouraging results for the use of synthetic data derived from CoNaLa Mined to aid in the task of translating natural language to code, irrespective of the downstream architecture or objective that uses it. This directly supports Hypothesis 2.3, that *Language models can be used to generate weak supervision data to improve the effectiveness of NL to code translation systems*.

5.5.3 Impact of Source Code Origin on Synthetic Data Effectiveness

While the previous section demonstrated that generating synthetic NL descriptions for code (as done with AugGPT3 for CoNaLa Mined) is beneficial, the characteristics of the source code itself can significantly influence the quality and utility of the resulting synthetic parallel corpus. This section investigates how the choice of source code for synthetic data generation impacts downstream model performance.

Methodology and Experimental Setup

To analyse the impact of the source code's origin, we first characterise several Python code corpora by their syntactic completeness, as measured by their ability to be parsed into an Abstract Syntax Tree (AST).

Source Code Datasets We consider four distinct Python datasets, whose parsing rates, shown in Table 5.8, reveal significant structural differences:

- **Hearthstone & CodeSearchNet:** These datasets have the highest parsing rates (95.1% and 96.9%, respectively) as they consist of well-formed, complete Python classes and functions.
- **CoNaLa (Mined):** Sourced from Stack Overflow, this dataset contains mostly complete, self-contained snippets, resulting in a high parsing rate of 92.3%.
- **Django:** This dataset has the lowest parsing rate (60.6%). Its samples are single lines extracted from the Django framework's source code, which often represent syntactically incomplete fragments (e.g., `if` conditions, return statements) that are not parsable in isolation.

Source	Parses	Number of samples
Hearthstone	95.1%	533
Django	60.6%	16,000
CoNaLa (Mined)	92.3%	50,000
CodeSearchNet	96.9%	184,916

Table 5.8: Parsing rates of target code from various Python source code collections.

Experimental Design To investigate how these structural differences affect the utility of synthetic data, we extend the AugGPT3 methodology. We use GPT-3 to generate synthetic English descriptions for code sampled from three of these sources: Hearthstone, Django, and the CoNaLa Mined set. This results in three new synthetic datasets, which we refer to as Synth Hearthstone, Synth Django, and Synth CoNaLa, respectively.

We then evaluate the effectiveness of these synthetic datasets by fine-tuning our BART > PreCode model under two distinct conditions:

1. **Unsupervised:** The model is fine-tuned *only* on one of the synthetic datasets (e.g., Synth Hearthstone) and then evaluated directly on the CoNaLa test set. This setup measures the direct transferability of knowledge learned from the synthetic data, testing how well it generalises to the target domain without seeing any of the target domain's training data.
2. **Transfer Learning:** The model is first fine-tuned on a synthetic dataset and then undergoes a second stage of fine-tuning on the standard CoNaLa training set before evaluation. This setup assesses whether the synthetic data provides a beneficial pre-training or initialisation step for subsequent in-domain fine-tuning.

The performance of these models is compared against a baseline BART > PreCode model fine-tuned only on the CoNaLa training set. All models in this experiment are trained using the translation objective, and performance is measured by BLEU and parsing accuracy on the CoNaLa test set.

Results and Analysis

Table 5.9 presents the results of our experiments. The findings highlight that the distributional similarity between the synthetic data's source code and the target task's code is a critical factor for success.

In the unsupervised setting, where models are trained only on synthetic data, the results show a stark contrast based on the data source. The model trained on Synth Hearthstone performs extremely poorly (8.36 BLEU, 3.6% Parses). The class-based structure of Hearthstone code is distributionally very different from the procedural snippets in CoNaLa. When prompted with a CoNaLa-style NL description, the model attempts to generate code in a Hearthstone-like class structure, leading to severe hallucinations and unparsable outputs, as detailed in the example in

Training Setup	**Source for Synth. NL**	**Method**	**BLEU**	**Parses**
Unsupervised	Hearthstone	BART > PreCode > Synth Hearthstone	8.36	3.6%
	Django	BART > PreCode > Synth Django	11.45	20.3%
	CoNaLa (Mined)	BART > PreCode > Synth CoNaLa	31.10	85.2%
Transfer learning	Hearthstone	BART > PreCode > Synth Hearthstone > CoNaLa Train	30.98	83.9%
	Django	BART > PreCode > Synth Django > CoNaLa Train	29.67	86.3%
	CoNaLa (Mined)	BART > PreCode > Synth CoNaLa > CoNaLa Train	33.32	94.0%
Fine-tune (Baseline)	—	BART > PreCode > CoNaLa Train	32.26	94.5%

Table 5.9: Comparison of source code origin for synthetic English translations in supervised and unsupervised settings. Models are fine-tuned using a translation objective. BLEU and Parsing scores are evaluated on the CoNaLa test set.

Listing 5.1. Similarly, the Synth Django model performs poorly (20.3% Parses), likely because it learns to generate the syntactically incomplete fragments characteristic of the Django dataset, which are invalid for the standalone snippets expected in the CoNaLa task. In stark contrast, the model trained on Synth CoNaLa (Mined) performs remarkably well (31.10 BLEU), nearly matching the baseline fine-tuned on the actual CoNaLa training data. This success is due to the close distributional similarity between the code in the CoNaLa Mined set and the CoNaLa test set, demonstrating that direct transfer is possible when the source and target distributions are well-aligned.

In the transfer learning setting, where models are further fine-tuned on the CoNaLa training set, performance significantly improves for all models. The high parsing rates indicate that the in-domain fine-tuning stage effectively adapts the models to the target domain's syntax and structure, correcting for the initial distributional mismatch. However, for the Hearthstone and Django-based models, the initial synthetic pre-training provided no discernible benefit over the baseline; their final scores are lower than the model trained directly on CoNaLa. This suggests that if the synthetic data's source is too dissimilar, it does not serve as a useful initialisation. The best overall performance is achieved by the model pre-trained on Synth CoNaLa (Mined) and then fine-tuned on CoNaLa Train (33.32 BLEU). This result, which slightly surpasses the baseline, reinforces that in-distribution synthetic data can provide a valuable, albeit small, performance boost.

In summary, these experiments underscore a key finding: while LLMs can generate plausible NL descriptions for diverse code, the utility of the resulting synthetic data is highly contingent on the distributional alignment between the synthetic data's source code and the target task. For direct, unsupervised application, this alignment is crucial. If a significant gap exists, robust in-domain fine-tuning is necessary to bridge it, and poorly chosen synthetic data may offer no advantage at all.

```
Input (CoNaLa prompt):
Q: "How can I send a signal from a Python program?"

Ground truth: os.kill(os.getpid(), signal.SIGUSR1)

--- Hallucinated output (bad, uncompilable code) ---
class SignalSender(SpellCard):
    def __init__(self):
        super().__init__("Signal Sender", 3, CHARACTER_CLASS.MAGE, CARD_RARITY.COMMON,
            target_func=signal.signal_target)

    def use(self, player, game, signal.SIGUSR1):
        super().use(player, game)

        os.signal_send(player.id, self.target.heal(signal.SIGKILL)) # Error: os.signal_send
            invalid API; heal(SIGKILL) semantic mess

        player.buff += "signal_boost"
        self.target.taunt = signal.signal(signal.SIGSTOP, self.handle_signal)

    def handle_signal(self, signum, frame):
        game.display_message("Signal sent: ", signum)
```

Listing 5.1: Hallucinated output from model trained on ‘Synth Hearthstone‘ when given a CoNaLa-style prompt.

Breakdown of errors in the hallucinated code shown in Listing 5.1:

Error Type	**Example Line / Fragment**	**Likely Dataset Origin / Explanation**
Invalid class wrapping	‘class SignalSender(SpellCard):‘	Hearthstone (Wraps procedural CoNaLa task in SpellCard class)
Fake ‘target_func‘	‘target$_{f}unc = signal.signal\_target$‘	Hearthstone + CodeSearchNet (Hallucinates nonexistent param)
Invalid function params	‘signal.SIGUSR1‘ (as param)	CodeSearchNet (Misuses constant as a parameter)
Invalid function call	‘os.signal_send(...)‘	CodeSearchNet (Hallucinates nonexistent API)
Semantic mismatch	‘self.target.heal(signal.SIGKILL)‘	Hearthstone (Attempts to "heal" with a ‘SIGKILL‘ constant)
Type error	‘player.buff += "signal_boost"‘	Hearthstone (Invalid string concat on likely non-string attribute)
API misuse	‘self.target.taunt = signal.signal(...)‘	Mixed (Randomly combines incompatible patterns)
Undefined variable	‘game.display_message(...)‘	Hearthstone (Calls method on ‘game‘, undefined in this scope)

5.6 Conclusion

We started this chapter with the aim of validating Hypothesis 2: *NL to code translation can be improved through sources of weak supervision*. In order to tackle this, we first proved that aligning the pre-training and fine-tuning objectives can be beneficial in a single language domain of conversational query rewriting. This set the stage for us to test this alignment with English and code, where we observed improvements in semantic similarity through the use of reiteration for BART models. We then continued the pre-training phase of the BART model with unsupervised data of joint English and code in the form of functions with their docstrings. Through the use of ablations, we confirmed that unsupervised data can have significant performance gains for both semantic similarity and syntactic correctness, and that these gains are aligned with the pre-training modality. Finally, we turned to language models solely trained on unsupervised corpora as a means to generate English descriptions from known code snippets. By fine-tuning BART models and specialised architectures on this synthetic data, we confirmed that it is effective in providing NL to code models with additional signal, often to the point where our previously discovered alignment can become obsolete. Our further investigation into using different source code corpora for synthetic data generation revealed the critical importance of distributional alignment between the source of the synthetic data and the target task. While synthetic data from closely related distributions (like CoNaLa Mined for the CoNaLa task) can be highly effective, data from disparate distributions (like Hearthstone) can lead to poor performance if not followed by substantial in-domain fine-tuning. Armed with these findings, we can confidently validate Hypothesis 2, with the added nuance that the choice and application of weak supervision sources require careful consideration of data characteristics and training regimes.

5.7 Broader Context and Subsequent Developments

The finding that infilling objectives are superior to direct translation for natural language to code translation has since been explored at scale in code language model training. The fill-in-the-middle (FIM) objective, introduced explicitly as a training technique and subsequently adopted in StarCoder [82] and Code Llama [116], is essentially the infilling principle studied in this chapter applied at scale with far larger models and corpora. These systems demonstrated that bidirectional context improves code completion, validating Hypothesis 2.1 as a durable finding rather than a dataset-specific observation. The additional complexity that FIM introduces to training and inference pipelines has since led many systems to favour standard causal language modelling objectives instead.

The use of language models to generate synthetic supervision data, studied in this chapter in the context of NL-to-code translation, has been validated on a substantially larger scale across NLP more broadly. The practice of generating synthetic data from capable language models and using it to train smaller ones has become a dominant paradigm in model alignment and fine-tuning, applied both to instruction following and to reasoning, where frontier models generate synthetic reasoning traces that are then distilled into smaller models [31].

The joint NL-and-code pre-training finding in Hypothesis 2.2 is now so thoroughly confirmed that it can barely be called a finding: every contemporary code language model trains on a mixture of natural language and code. As noted in the introduction, the reasoning improvements seen when code was added to the Llama training mixture directly demonstrate this at scale [130, 131].

Chapter 6

Query-Specific Programs for Tabular Question Answering

This work was carried out in 2021–2023 and published at EMNLP 2023 [44]. Large language models had demonstrated strong language understanding and generation, but were reliably failing at the precise arithmetic and structured reasoning that data-intensive tasks require [98]. The dominant response in tabular question answering was to linearise tables into longer token sequences, following a broader trend in which context windows had grown from BERT's 512-token limit [32] to the 2048-token capacity of GPT-3 [16]. Whether models could make productive use of this expanded context for structured data reasoning was, however, untested.

Having established effective methods for language models to generate code from natural language descriptions in the preceding chapters (Hypotheses 1 and 2), this chapter now investigates how these code-generation capabilities can be leveraged for tool use, specifically in the context of tabular question answering (Hypothesis 3). We will build on the code-generating abilities discussed and apply them as an intermediary stage within this task. We will first describe the task and the existing approaches to solving the problem, followed by their shortcomings. We will then test the effect that filters in the form of short programs can have on improving their effectiveness. Finally, we will develop a pipeline that generates these programs dynamically and study its effect on some of the most challenging samples in tabular question answering. To this end, we aim to validate the following overall Hypothesis: *Language models can generate question-specific programs to improve question answering accuracy from tabular data*.

To put the contributions of this chapter in the broader context of this thesis, Figure 6.1 describes all components used in this chapter within the tool use framework presented in Chapter 2. This is the first time in this thesis where we close the tool use loop and augment language models with executable programs.

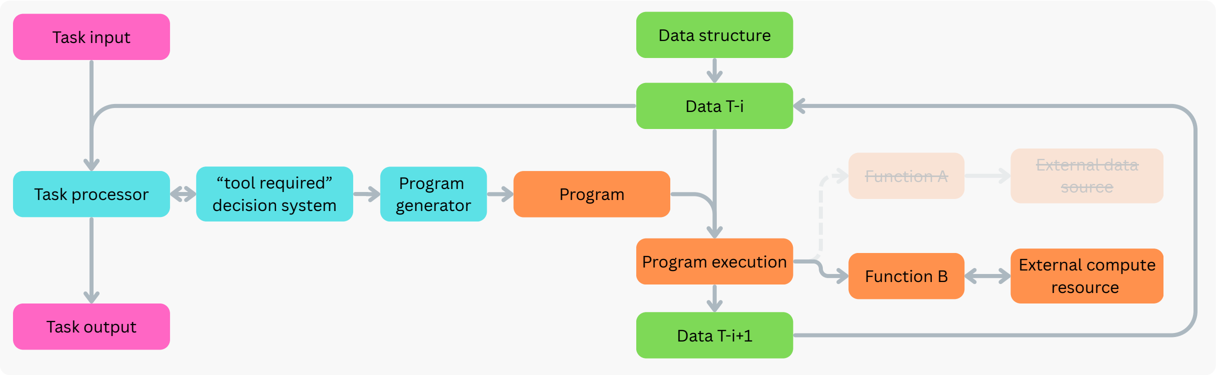


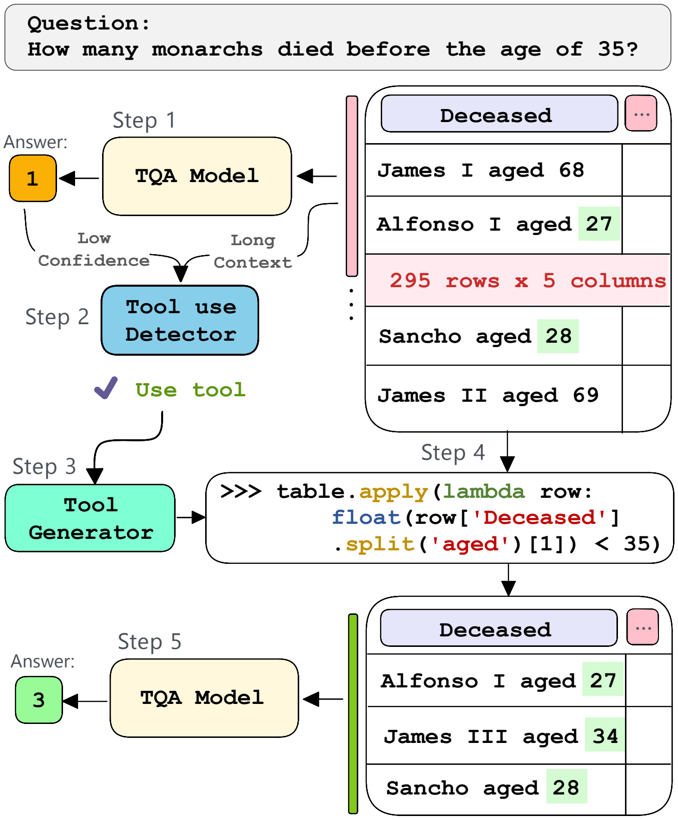
Figure 6.1: The tool use diagram showing the first time in this thesis where we close the tool use loop. **At the core of tabular question answering (TQA) is a table, a semi-structured data structure allowing for scalable operations like row-filtering. Data T_i contains a table instantiation for a given sample.** **For a given table, the task is to produce an exact answer to a natural language question. Both task input and output are represented as text.** **A fully neural TQA system like BART acts as the task processor, byproducts of inference like sequence log probability are used as the "tool required" decision system. To use the tool and generate the row-filtering program, a distinct language model like T5 is used.** **The generated program is executed on data T_i making use of dedicated library functions to produce a transformed table representing data T_{i+1}. No external data sources are used, and such are hidden in the diagram.** The collection of all components working together is what, in this work, we call ToolWriter.

6.1 Introduction

Recent research has focused on expanding the capabilities of large language models like T5, PaLM, and GPT-3 through integration with external tools and APIs [42, 78, 104, 119]. While these tool-augmented systems allow models to access external knowledge and computational resources, they have a limited ability to handle semi-structured data formats. This limitation becomes particularly apparent when dealing with complex questions over large datasets, where model performance tends to degrade significantly. One domain where these challenges are especially evident is tabular question answering (TQA), as the combination of lengthy tables and intricate questions poses substantial difficulties for existing approaches. This highlights a key limitation of the language model's processing: while adept at understanding the query and general table semantics, it struggles with the precise, scalable processing required by large, structured inputs, a task better suited for dedicated digital systems.

The field of tabular question answering focuses on extracting and reasoning over information contained in semi-structured tables to respond to complex, multi-step questions. The presence of natural language elements embedded within the tabular structure makes purely programmatic solutions inadequate. This can be observed in Figure 6.2, where table cells may contain numbers that derive meaning from their surrounding textual context, such as the qualifier "aged".

Contemporary approaches to tabular question answering, including systems like Unified-



```
>>> table.apply(lambda row:
        float(row['Deceased']
        .split('aged')[1]) < 35)
```

Figure 6.2: ToolWriter for tabular question answering introduces: 1) A tool use detector; 2) A tool generator. Here, a row-filter tool is generated as a program that transforms semi-structured data.

SKG, OmniTab, and TapEx [64, 88, 145], convert tables into sequential token strings for joint processing with questions. To handle the extensive tables found in datasets like WikiTableQuestions [106] and WikiSQL [156], these systems employ language models with expanded context windows of 1024 tokens, despite the substantial computational overhead this entails. Our analysis in Section 6.3 reveals two critical limitations: a substantial 40% decline in model effectiveness as tables grow in size, and the inability to fully process 23% of WikiTableQuestions tables due to the 1024-token constraint.

To address these limitations, we introduce ToolWriter, an approach designed to overcome these limitations by interleaving program generation and neural generation. ToolWriter employs a two-stage process: first, determining whether tool assistance is necessary for a given table-question pair, then dynamically generating a specialised program, specifically a row filter to perform the precise, structured data manipulation that the language model struggles with, thereby simplifying the input for the language model's subsequent interpretation. A key advantage of our approach is that the generated programs can process tables of any size, effectively circumventing the context limitations of language models. Furthermore, ToolWriter's architecture is designed to be model-independent, allowing integration with various existing systems through zero-shot deployment.

In this work, we compare multiple detection strategies and tool generation approaches to generate query and table-specific Python programs. Our system, ToolWriter, combines zero-shot GPT-3 for tool generation with a hybrid detection mechanism based on answer confidence and table length. This approach yields significant performance improvements, achieving exact match scores of 64.9% (+1.9%) on WikiTableQuestions and 90.5% (+1.5%) on WikiSQL.

The key contributions of this research are threefold:

- A comprehensive analysis of language model performance on tabular question answering on WikiTableQuestions, revealing systematic degradation correlated with increasing table size.
- The introduction of ToolWriter, a novel framework that dynamically identifies tool requirements and generates specialised data transformation programs. This approach establishes new performance benchmarks on both WikiTableQuestions and WikiSQL datasets.
- Empirical validation through ablation studies demonstrating that our tool generation methodology is architecture-independent and exhibits enhanced performance on larger tables.

6.2 Task Definition

In tabular question answering, systems must extract information from structured tables T to produce answers $\hat{y}$ to natural language questions q. The task requires both understanding natural language and performing multi-step reasoning over symbolic data to combine and analyse the table appropriately.

The task provides a dataset D containing triplets of questions, tables, and ground truth answers: $D = \{(x = (q, T), \hat{y})\}$. To evaluate generalisation, tables in training and test sets are non-overlapping. Questions must be answerable using only information present in the corresponding table. System performance is measured using exact match accuracy (EM) between predicted answers $\hat{y}$ and ground truth y averaged over D.

6.2.1 Datasets

WikiTableQuestions [106] provides a challenging benchmark for evaluating model limitations. This dataset contains 2,108 HTML tables with crowdsourced question-answer pairs. While multiple questions exist per table, the training and test sets use separate tables. The dataset presents several unique challenges:

- Questions require multi-step reasoning to combine information from different parts of a table
- Table formatting is inconsistent, with cell values that rely on reader interpretation

- Mixed natural language and structured data within cells makes pure programmatic solutions difficult

WikiTableQuestions-Filter represents a focused subset of the WikiTableQuestions dev set used to evaluate ToolWriter's filtering capabilities in Section 6.5.1. Using SQUALL annotations [125], we identify 1256 samples containing explicit filtering operations through `WHERE` clauses following `SELECT` statements.

WikiSQL [156] encompasses 80,654 question-answer pairs across 24,241 Wikipedia tables. While originally designed for semantic parsing, it now serves as a weak supervision benchmark using answer spans as ground truth. Its tables have well-defined types and simpler questions compared to WikiTableQuestions, with operations limited to SQL-parseable cell values.

6.2.2 Baseline models

We examine the capabilities of systems trained with various degrees of task-specific supervision.

BART [81], built on the Transformer architecture [133], employs denoising pre-training. It processes TQA tasks by combining queries and linearised tables within a 1024 token window using special tokens: $x = q\,\texttt{[HEAD]},c1,,cN,\,\texttt{[ROW]},1,r1,\,\texttt{[ROW]},2,r2$.

TapEx [88] extends BART through SQL execution training on synthetic data, achieving leading results on WikiSQL's weak supervision benchmark.

OmniTab [64] builds upon TapEx with natural language fine-tuning, converting SQL to questions and leveraging table-aligned text. It leads WikiTableQuestions performance through specialised training.

UnifiedSKG [145] adapts T5 [114] for structured knowledge tasks using a unified text-to-text approach across diverse data types (tables, knowledge bases, semantic parsing, etc.).

FlanT5 [135] balances specialisation and generality through instruction-tuning across 62 NLP tasks, establishing strong TQA performance via in-context learning.

GPT-3 [16] demonstrates unsupervised TQA capabilities through zero-shot chain-of-thought reasoning [21, 73, 137], providing step-by-step solutions.

6.3 Behaviour Analysis

A key challenge in tabular question answering stems from the extensive size of input tables. Due to computational constraints, transformer models typically have a context window of 512 tokens [133]. Analysis of WikiTableQuestions reveals that 41.7% of tables surpass this limit before including the size of the question text. While some systems expand this window to 1024 or 2048 tokens, this approach leads to substantial memory overhead and still results in data truncation for 23.8% of cases at the 1024 token threshold.

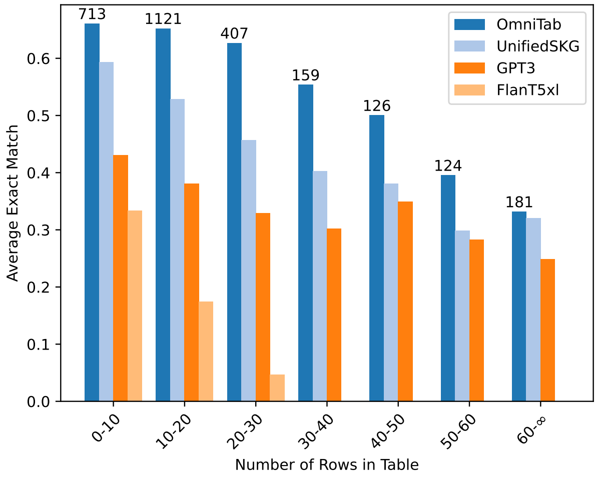


Figure 6.3: Exact match by table size on the WikiTableQuestions dev set. The number of dataset samples per row subset is shown above each bar.

6.3.1 Effectiveness across table size

Our analysis in Figure 6.3 examines model performance across varying table sizes in WikiTableQuestions dev. The data shows that performance deteriorates significantly before reaching the 40-50 row range, where context limits become a factor. This suggests table length impacts effectiveness beyond just context window constraints. This performance degradation highlights a key limitation of the language model's capabilities: while adept at understanding the query and general table semantics, it struggles with the precise, scalable processing (like filtering or aggregation) required by large, structured inputs, a task better suited for symbolic programs.

The impact becomes particularly pronounced for tables with over 50 rows, where accuracy drops 40% compared to smaller tables. While some models maintain limited functionality by leveraging information in the initial rows, we find that FlanT5-XL exhibits severe degradation, often manifesting as repetitive token hallucinations when processing larger tables. To further complement the analysis of input length on model performance, in Table B.1 we perform further ablations on the SQUALL annotated data to characterise model performance on different types of reasoning required for tabular question answering.

6.3.2 Potential in row filtering

Given our observation that table size negatively impacts model performance, we investigate strategies for filtering irrelevant data. We propose that selective row filtering could enhance accuracy by reducing noise while preserving essential information. To evaluate this, we implement a manual row filtering approach that retains only question-relevant rows.

We conduct an experiment on 51 samples from the WikiTableQuestions dev set, comparing targeted filtering against random 50% row removal. Input formatting remains consistent across conditions, varying only in row count. Table 6.1 presents our analysis of selective versus random

Row Filter	U.SKG	O.Tab	GPT-3	FlanT5-XL
None	39.21	54.9	47.05	17.64
Random	11.76	5.88	15.64	5.52
Manual	**60.78**	**58.82**	**54.9**	**39.21**

Table 6.1: Exact match scores over 51 samples from the WikiTableQuestions dev set with gold rows selected manually as sufficient to answer the question.

filtering, revealing two key insights:

Correct row filtering enhances model performance substantially. Reducing table size while maintaining critical data allows complete processing of 4 previously truncated tables. This targeted approach corrects 10% of previously incorrect predictions by eliminating distracting information.

Incorrect usage of a tool poses the risk of removing relevant information. The discrete nature of tabular data means models struggle to compensate for missing information, as demonstrated by poor random filtering results. Additionally, we find 7 cases where filtering proves unnecessary, such as for questions like *"how many players participated?"*. This highlights the need for detection strategies for questions that don't demand filtering.

These results demonstrate the need for an automated system capable of generating targeted filtering operations while intelligently determining when to apply them. As a result, we validate Hypothesis 3.1: *Performance of language-model-based TQA systems is influenced by table size and can be improved by selectively filtering tabular data inputs.*

6.4 ToolWriter

ToolWriter is our proposed solution to overcome the challenges faced by current language models when dealing with large semi-structured datasets. When a question is posed about a table, ToolWriter evaluates the necessity of using a tool. It then creates a program to modify and streamline the table, enhancing the effectiveness of question-answering. We begin by presenting our conceptual framework, followed by the implementation of our method.

6.4.1 Proposed Model

We present ToolWriter as TW, which integrates a $Tool$ with a TQA model F, controlled by $\sigma \in [0,1]$, where 0 indicates "do not apply tool" and 1 indicates "apply tool".

$$TW(x,F) = \sigma \cdot F(Tool(x)) + (1-\sigma) \cdot F(x) \tag{6.1}$$

Furthermore, we define σ as the result of a *detector* function that estimates the uncertainty of the model's prediction.

$$\sigma = d_\theta(x, F(x)) \approx P(y \neq F(x)) \tag{6.2}$$

Our objective is to maximise the exact match *EM* across our dataset *D*. We now consider the partitions of *D* based on the correct and incorrect predictions of the model *F* (Eq. 6.3).

$$\begin{aligned} \overline{S_F} &= \{(x,\hat{y}) \in D | F(x) \neq \hat{y}\} \\ S_F &= \{(x,\hat{y}) \in D | F(x) = \hat{y}\} \end{aligned} \tag{6.3}$$

As intended, transforming an input *x* with a tool can have either a beneficial or detrimental impact on the output. By examining our two subsets of *D*, we can draw the following conclusion.

$$EM(TW(\cdot,F),\overline{S_F}) = \begin{cases} 0 & \sigma = 0 \\ \leq 1 & \sigma = 1 \end{cases} \tag{6.4}$$

$$EM(TW(\cdot,F),S_F) = \begin{cases} 1 & \sigma = 0 \\ \leq 1 & \sigma = 1 \end{cases} \tag{6.5}$$

When the model makes an incorrect prediction, applying the tool transformation to *x* will either improve or maintain performance (Eq 6.4), while it risks degrading performance when the model is already correct (Eq 6.5). This motivates using a detector function that estimates model error probability, as defined in Eq. 6.2.

Tool Use Detection

To optimise *EM* across both subsets S_F and $\overline{S_F}$, we use σ to control tool application. While the original prediction is optimal for S_F, applying our tool before model *F* can improve results on $\overline{S_F}$. With access to ground truth answers, we could construct a perfect oracle detector for tool usage.

In real scenarios, we employ a parametrised detector *d* to approximate the oracle. This builds on established research in query difficulty estimation [30] and performance prediction.

Section 6.4.2 provides details on our parametrised detector implementation. Setting $\sigma = 1$ represents constant tool application. Since we aim to detect model failure cases, our tool design critically influences performance on the target subset of *D*.

Programs as Tools

By definition, $\overline{S_F}$ contains examples challenging for *F*. Our experiments in Section 6.3 reveal consistent failure patterns across various state-of-the-art neural models (different versions of *F*).

This prompts a key question: **What tool can distinctly complement the learned abilities of neural models?**

We hypothesise that programs offer two key advantages: 1) provide a natural interface to structured data; 2) circumvent several innate limitations of current neural systems due to their extrapolative nature. Our tools framework transforms input data x to modified input x^* within domain I, with potential noise ε during transformation, as shown in Eq 6.6:

$$Tool : I \rightarrow I^* + \varepsilon \tag{6.6}$$

ToolWriter combines program generation with a detection mechanism d to enhance model performance on challenging inputs. A program generator $C(x)$ produces input-specific Python code, which is then executed on the input:

$$Tool(x) = Exec[x, C(x)] \tag{6.7}$$

The choice of program tools depends heavily on both the specific task requirements and the identified model weaknesses. We detail our task formulation in Section 6.2, followed by an in-depth analysis to guide our search for useful programs as tools.

6.4.2 Model implementation

ToolWriter integrates a detection module and program generator with existing TQA models. Based on our analysis, we employ OmniTab and UnifiedSKG as base models $F(x)$ for WikiTableQuestions, while using TapEx for WikiSQL.

Model-agnostic Tool use Detector

We develop a model-agnostic detector $d_\theta(x, F(x))$ to detect when a tool is likely to improve model accuracy. Detecting input difficulty is a reasonable assumption and aligns with previous work on quality estimation [39, 132] and query performance prediction [30]. The **combined** detector in ToolWriter is a linear classifier with the following features:

Sequence log-probability (SeqLogProb) is the length-normalised sequence log-probability from a trained model $F(y \mid y, x, \theta)$.

$$\frac{1}{L} \sum_{k=1}^{L} \log F(y_k \mid y_{<k}, x, \theta) \tag{6.8}$$

We expect low-confidence answers are likely to be incorrect.

Input length, as we have seen in Section 6.3, poses a challenge to all models irrespective of size and training objective. We leverage the size of the table, measured by the number of rows, as a simple feature to decide when to apply a tool.

Our use of such simple detection methods contrasts with well-studied error detection methods in NLP [8,39] as a sign that tools are reasonable model-agnostic extensions even with simple detection heuristics.

Row-filter Tool Generator

The tool generator synthesises a short Python program that takes a table as input and returns a transformed version of it. The following code snippet is fixed and highlights the area where the generated code from the tool generator is placed.

```
new_table = table[table.apply(lambda ... , axis=1)]
```

The task of the row filter generator is to generate a lambda function to remove the rows in the table that are not relevant to answering the question. The program may be of arbitrary complexity, emphasising the generality of our approach for systems to interact with data through programs as seen in Figure 6.4.

As observed in Table 6.1, removing rows requires care to preserve crucial information. It is evident that a tool must adapt its filtering strategy according to the question and the table. Our row filtering tool keeps its input and output space consistent and suitable for the downstream model.

The task of generating tools requires the model to produce an explicit transformation of the table given the question. Given that the search space for tools as programs grows exponentially with the expressiveness of the tools, we opt for methods that reduce the search space by having a prior on what possible transformations will work best.

Specifically, we explore two approaches for generating Python row filters:

Fine-tuned T5. We fine-tune a T5 model through supervised training to auto-regressively generate a Python row filter given a question q and a table T. For our supervised data, we leverage a subset of WikiTableQuestions with SQUALL [125] annotations on questions that contain a single `SELECT` and `WHERE` clause, which are likely to benefit from row filtering. Our starting checkpoint is FlanT5-XL, using a batch size of 64 on two RTX 3090 GPUs for 10k steps for 8 hours.

Zero-shot GPT-3. We leverage GPT-3 for zero-shot prediction to generate a row filter as a Python lambda function. We use the "text-davinci-003" API with a temperature of 0.2 with the question and table schema in the prompt (Figure 6.5). Zero-shot tool generation shows the potential of low-effort approaches to manipulate structured data.

```
# Extracts x from “x-y” and keep if greater than 2
lambda row: float(row['Score'].split(‘-’)[0]) >= 2
```

(a) Zero-shot GPT-3 for the question "In how many games did sri lanka score at least 2 goals?"

```
# Keep rows containing ‘France’ in the description
lambda row: 'france' in row['Description'].lower()
```

(b) T5 for the question "Is France mentioned positively?"

Figure 6.4: Example row filter tools generated from our two proposed methods. Comments are added manually for explanatory purposes.

Tool	**Detector**	**OmniTab**	**UnifiedSKG**
Baseline		73.5	55.5
T5	**Always**	72.0 (-1.4)	59.5 (+4.0)
	Oracle	77.5 (+4.0)	63.3 (+7.8)
GPT-3	**Always**	74.6 (+1.1)	63.0 (+7.6)
	Oracle	80.3 (+6.8)	68.6 (+13.1)
Human SQL	**Always**	74.9 (+1.4)	65.0 (+9.5)
	Oracle	82.7 (+9.2)	70.6 (+15.1)

Table 6.2: Row filter tool performance on WikiTableQuestions-Filtered with two detection strategies.

Figure 6.4 showcases generated Python code samples from both our proposed tool generators. Language models have no formal guarantees for executable code [20, 113]. As a result, if the execution of the tool throws an exception or the resulting table is empty, we revert to the original table.

Manual tool. For WikiTableQuestions-Filter, we leverage the SQUALL SQL annotations to derive a manual row filter. We analyse row filter headroom performance in Section 6.5.1.

6.5 Results

First, we investigate the various tool generators of ToolWriter on a subset of WikiTableQuestions where filtering is often required. Second, we focus on the importance of the detector to choose when to best apply the generated tools (Section 6.5.2). Third, we test ToolWriter to both detect and generate tools across various TQA datasets and methods (Section 6.5.3). Finally, we analyse how ToolWriter performs as the table size increases (Section 6.5.4).

6.5.1 Performance of tool generators

As we observe in Table 6.2, our automatic tool generators (T5, zero-shot GPT-3) almost universally increase model performance on WikiTableQuestions-Filtered. Importantly, regardless of how our tools may be applied, UnifiedSKG significantly benefits from all tools generated by ToolWriter. This shows our tools are effective at filtering irrelevant information from tables that would otherwise cause TQA models to fail. These findings prove the validity of Hypoth-

```
User 1:
I need an expert to help me answer the question by making the table
smaller.
Question: Who are all of the players on the Westchester High School
club team?

table = {'Player': ['Jarrett Jack', 'Jermaine Jackson', ...
'No.': ['1', '8', ...
'Nationality': ['United States', 'United States', ...
'Position': ['Guard', 'Guard', ...
'Years in Toronto': ['2009-10', '2002-03', ...
'School/Club Team': ['Georgia Tech', 'Detroit', ...
}

User 2:
For 'Who are all of the players on the Westchester High School club
team?' the most impactful change will be to filter the rows. Since I
don't know all the rows I'll use rough string matching, float
   casting,
lowering and be as broad as possible.

>>> new_table = table[table.apply(lambda
   row_dict:|\colorbox{magenta!30}{ }|'Westchester' in
row_dict['School/Club Team'].lower(), axis=1)]
```

Figure 6.5: Prompt used to generate row filter tools with GPT-3 in a zero-shot setup. Tables are truncated to two rows to give the model a schema for how to interact with the data. The highlighted region indicates the start of the prompt completion.

	WikiTableQuestions				**WikiSQL**	
	OmniTab		**UnifiedSKG**		**TapEx**	
Detector	Dev	Test	Dev	Test	Dev	Test
Never apply	62.7	63.0	49.6	50.8	89.6	89.0
Always apply	56.5 (-6.2)	57.5 (-5.5)	48.5 (-1.0)	50.2 (-0.6)	89.6 (0.0)	89.8 (+0.7)
SeqLogProb	63.7 (+1.0)	64.3 (+1.2)	52.6 (+3.0)	54.6 (+3.8)	90.5 (+0.8)	90.4 (+1.3)
Combined	63.7 (+1.0)	64.9 (+1.8)	52.9 (+3.4)	54.5 (+3.7)	90.7 (+1.1)	90.5 (+1.5)
Oracle	67.4 (+4.7)	68.3 (+5.3)	57.9 (+8.3)	59.0 (+8.1)	91.7 (+2.0)	91.5 (+2.4)

Table 6.3: Exact match results on various detection strategies for applying our best row-filter tool generator: GPT-3.

esis 3.2: *In TQA settings when filter operations are applicable, language models can generate query-specific programs that, when executed, improve the accuracy of neural TQA systems*.

Manual tools show the potential for tool generators to simplify tables further. Our best tool generator, zero-shot GPT-3, achieves 70% of manual performance averaged over all detection settings and models.

Table 6.2 also highlights the importance of the detector. We observe a large performance gap for all tool generators when comparing the ’Always’ application strategy with the ’Oracle’ detector.

6.5.2 Detecting when to use tools

Table 6.3 shows the performance of multiple detection strategies on the full dev and test sets for WikiTableQuestions and WikiSQL. We use our best-performing tool generator, zero-shot GPT-3. We observe row filtering tools require query-specific detection since "always" or "never" applying tools shows the lowest results in all cases.

We observe that even simple detection methods like SeqLogProb are sufficient to inform ToolWriter when to apply the query-specific generated row filter. We see significant benefits in leveraging tools for all TQA models in contrast to not using them. Performance increases further as we include table length as a feature in our detector, highlighting the importance of using tools in accordance with the complexity of the data. With these findings in mind, we can validate Hypothesis 3.3: *Language model signals can be used to identify when to apply query-specific programs automatically, thus leading to improved accuracy in neural TQA systems*.

Under the oracle detection conditions, we observe significant potential for our generated tools. This shows how deciding **when** to apply a row filter tool is just as important as **how** to apply it.

Method	**Dev**	**Test**
2-shot GPT-3 Direct [21]	—	27.3
BART [81]	37.2	38.0
2-shot GPT-3 CoT [21]	—	45.7
UnifiedSKG [145]	50.9 (49.6)	50.9 (50.8)
ToolWriter + UnifiedSKG	**52.9**	**54.5**
TapEx [88]	57.0	57.5
OmniTab [64]	— (62.7)	62.8 (63.0)
ToolWriter + OmniTab	**63.7**	**64.9**

Table 6.4: Exact match accuracy results on WikiTableQuestions. Results in parentheses are our reproduced experiments.

Method	**Dev**	**Test**
BART [81]	87.3	85.8
UnifiedSKG [145]	87.4	85.7
OmniTab [64]	—	88.7
TapEx [88]	89.2 (89.6)	89.5 (89.0)
ToolWriter + TapEx	**90.7**	**90.5**

Table 6.5: Exact match accuracy results on WikiSQL. Results in parentheses are our reproduced experiments.

6.5.3 Overall performance

Leveraging our prior findings, ToolWriter is the combination of our best detection method (SeqLogProb with table length) and our best row-filter tool generator (zero-shot GPT-3). For each dataset, we show the corresponding model F as our base TQA model. Table 6.4 and Table 6.5 show overall model performance on WikiTableQuestions and WikiSQL, respectively.

Our results show that ToolWriter significantly improves performance agnostic of the target model using the generated tools. UnifiedSKG is particularly effective in leveraging the transformed tables with a 3.6% absolute performance increase compared to not using tools. When paired with OmniTab and TapEx, our method achieves new state-of-the-art results for both datasets. The improvement in WikiSQL is particularly impactful as ToolWriter enables a 10% error-rate reduction.

These results show how programmatic tools effectively complement neural components as an effective method for processing semi-structured data. In the following section, we perform a stratified analysis to understand where ToolWriter leads to the most improvements.

		OmniTab		**UnifiedSKG**	
		Dev	Test	Dev	Test
rows $<$ 30	**Baseline**	67.0	67.9	52.7	54.3
	ToolWriter	67.0 (+0.0)	**68.0 (+0.1)**	**55.0 (+2.3)**	**57.2 (+2.8)**
30 $\leq$ rows $<$ 60	**Baseline**	48.7	46.4	39.6	37.5
	ToolWriter	**52.6 (+3.9)**	**51.9 (+5.5)**	**44.0 (+4.4)**	**45.1 (+7.6)**
rows $\leq$ 60	**Baseline**	41.4	35.0	33.7	34.3
	ToolWriter	**48.1 (+6.6)**	**43.3 (+8.3)**	**42.0 (+8.3)**	**41.7 (+7.5)**

Table 6.6: Row filtering performance comparison on partitions stratified by table length for WikiTableQuestions.

6.5.4 Tools improve performance on long tables

In this section, we conduct an ablation study stratified by table length on WikiTableQuestions: short tables (rows $<$ 30), medium tables (30 $\leq$ rows $<$ 60), and long tables (60 $\leq$ rows). We aim to quantify the effect ToolWriter has as the table size increases. As in Section 6.5.3, ToolWriter uses GPT-3 as the tool generator and the combined detector.

Table 6.6 confirms our original hypothesis: Row filtering tools can be an effective strategy to help models handle long tables. We notice how, as the table length increases, the positive effect of the row filtering tool becomes more pronounced. Our Hypothesis is further confirmed with our T5 tool generator, where results mimic those in Table 6.6, reaching up to 5% absolute improvement with UnifiedSKG.

As noted in Section 6.4.2, detection is critical to tool use. On short tables, we observe no degradation in performance, highlighting the effectiveness of our combined detector.

6.6 Limitations

While ToolWriter enables language models to programmatically interact with tabular data, there are some clear limitations to what can be achieved with our method in its current form. One such limitation is the tool generator's focus on row filtering as a primary means to transform the table. ToolWriter relies exclusively on the underlying TQA model to perform all aggregations, arithmetic mappings, and groupings on the transformed table. This is because ToolWriter solely removes irrelevant rows from the input table, but may not reduce the size of the relevant rows, which, in the limit, may still overwhelm a Transformer-based TQA model. Our current approach also does not allow neural filtering of the tabular data for situations where programmatic interactions may break down. This is due to the current tool generator producing standard Python. Despite Python's expressive capabilities for string manipulation, as shown in Figure 6.4, string manipulations can break down in more semantic contexts [23]. Beyond this, we found that ToolWriter is sensitive to cascading errors due to the inherent multi-step interactions on the structured data. The tool generator is not informed by the failure mode of the original TQA model other than that it needs to reduce the table. This is due in part to the black-box nature of models such as Tapex which do TQA entirely end-to-end. Possible solutions to this include recent work in language model reasoning [17, 135] where a rationale may be explicitly generated, indicating a failure mode. This feedback would then inform the tool generation process.

6.7 Conclusion

Tabular question answering is a challenging setting for neural methods due to large table sizes and implicit reasoning. First, we characterise the limitations of neural methods to integrate structured data and find that all language modelling methods struggle with large tables. Second, we propose ToolWriter to generate query-specific tools to simplify large tables and detect when these transformations should be applied. We propose various language-model-based methods to generate programs that filter rows, which universally improve and achieve state-of-the-art results on two tabular question-answering datasets. Finally, we determine significant headroom in both detecting when to use tools and how to generate them under oracle settings, highlighting the potential in tools to manipulate structured data combined with language models. Overall, we can confidently state that *Language models can generate question-specific programs to improve question answering accuracy from tabular data*, thus validating Hypothesis 3.

6.8 Broader Context and Subsequent Developments

The central empirical finding of this chapter is that neural tabular question answering performance degrades as table size increases, and that this degradation occurs before any context-window truncation. This observation has since been independently confirmed at scale. Liu et al. [87] demonstrated that large language models struggle to retrieve and use information from the middle of long contexts, with performance degrading as input length grows regardless of whether the relevant content is nominally within the context window. Kate et al. [68] reached parallel conclusions in a code execution setting, finding that models fail on long-context function evaluation tasks in ways that mirror the structured-data degradation curves documented here. The chapter's table-size partitioning experiments (Table 6.6) provide an early, controlled characterisation of the same phenomenon that these subsequent studies identify as a general property of transformer-based language models on structured and semi-structured inputs.

As noted in the introduction, the program-as-tool pattern has since been adopted at the infrastructure level. The Model Context Protocol [3] standardises how language models discover and invoke external tools, and frontier reasoning models including o1, o3, DeepSeek-R1 [31], and Gemini embed Python execution directly within their reasoning traces. These systems address exactly the failure mode this chapter characterises: structured data is difficult for language models to reason over directly, and program execution provides a tractable alternative.

Chapter 7

Applications of Code-Augmented Language Models

This work was carried out in 2021–2022 and published at SIGDial 2022 [45, 46]. At that time, task-oriented dialogue systems were divided between two paradigms: symbolic approaches modelling conversation as explicit transition logic offered reliable grounding but scaled poorly as functionality grew, while neural language models produced fluent responses but could not maintain state across multi-turn interactions. The most capable language models were accessible only via commercial API and were unsuitable for real-time deployment in a voice agent, making lightweight, locally deployable sequence-to-sequence architectures a practical requirement. Using a sequence-to-sequence model to generate Python code as the action policy of a task-oriented agent, rather than predicting intent labels or filling slot values, was not an established design pattern.

This chapter demonstrates the practical, real-world application of the thesis's core arguments. It showcases how the principles of language model-driven code generation and tool use were implemented in GRILLBot, the winning entry in the Amazon Alexa Prize TaskBot competition. The Neural Decision Parser (NDP), a key component I developed, serves as the central case study. This chapter provides tangible evidence that augmenting language models with programmatic tools is an effective strategy for building robust, task-oriented AI systems.

This chapter fully applies the framework from Chapter 2 in a real-world multi-step system, instantiating its core components: a Task Processor, a "Tool Required" Detection System, a Program Generator, Data State (T_i and T_{i+1}), and Program Execution. We contextualise this application within the domain of task-oriented dialogue, which presents unique challenges not found in open-domain chit-chat. In this domain, we introduce an important component, the neural decision parser (NDP), that acts as the program generator in our tool use framework (see Figure 7.1). As is shown throughout this chapter, the NDP acts as a key component in contextualising user intent and translating it into action, as we've seen in Chapters 4 & 5. Despite this chapter's application focus, the successful modularisation of the NDP within GRILLBot

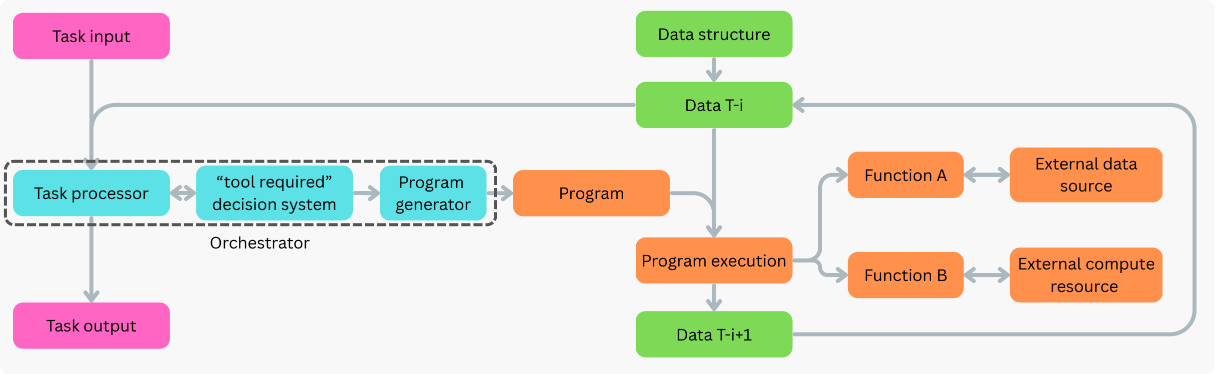


Figure 7.1: This thesis's tool use framework diagram with all components used in this chapter. **The task of GRILLBot is to continually respond to user utterances with the aim of completing a task. The user utterance is represented as text as well as GRILLBot's output. The Orchestrator acts as the agent in our framework, containing various neural components. An embedding-based intent classifier acts as a task processor and tool decision system, while the neural decision parser (NDP) acts as the program generator. Data structures like TaskGraphs and the task state act as placeholders for raw data crawled from the internet, and the current progress of the task is updated at runtime at each turn. The Python code generated by the NDP is executed on the data and calls external APIs and resources to answer task-related questions or access external knowledge.**

supports Hypothesis 1.1, which states that transformers are an effective method for the task of NL to code translation. Furthermore, this chapter provides application-focused evidence to this thesis's overall claim.

7.1 Introduction

Task-oriented dialogues involve long-horizon tasks with many intermediate steps requiring state persistence. The system must maintain the "thread" of the conversation over many turns while handling interruptions, questions, and conversational navigation (e.g., "let's go back to step 4"). This implicit structure demands a robust architecture that can separate concerns, allowing for rapid iteration—a critical factor in a competitive environment like the Alexa Prize. By creating a modular architecture, components such as new recipes or new question-answering sources can be added or improved in isolation without retraining the core logic.

This design philosophy is also motivated by the inherent limitations of parametric knowledge in language models. Language models have a knowledge cut-off determined by their training data, and while they memorise many facts, their parametric knowledge is an imperfect and unreliable database [110]. Editing this knowledge is difficult, often requiring full retraining or complex weight-patching methods [18] that may not even generalise compositionally; for instance, teaching a model that "Jack Depp is the son of Johnny Depp" does not automatically enable it to infer that "Jack Depp is the sibling of Lily-Rose Depp" [27]. Injecting updated information directly into the model's context is often more effective than attempting to edit its weights.

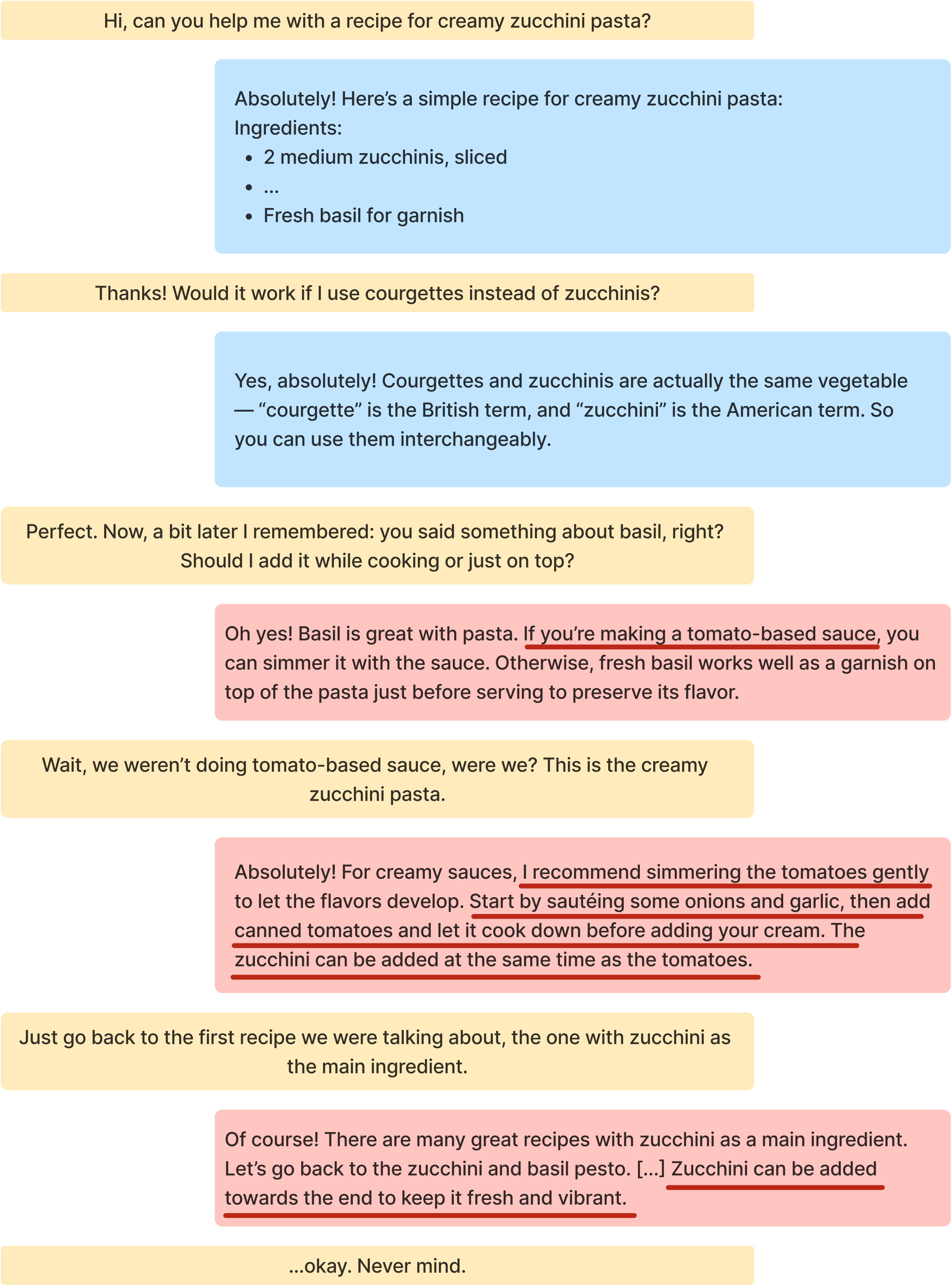
As covered in Chapter 3, conversational state management is key for coherent and engaging human-computer interaction. Traditional approaches often rely on structured methods like finite-state machines or intent-based dialogue trees [12]. An example of this would be the interpretation of the user utterance "Yes" mid-conversation. "Yes" can have various effects depending on the previous utterance: "Would you like to proceed with this recipe?", "Shall I set a timer for that?", "Do you want to skip this step?", etc. Intent trees model the transition logic in every part of a conversation. As the functionality increases, so does the size of the tree, often exponentially. While these methods offer predictability, they can be brittle, struggling to handle the variability and ambiguity inherent in natural human language, especially when user utterances deviate from predefined paths. Such systems, heavily reliant on matching predefined rules, like regex patterns, often lack the flexibility to adapt to out-of-distribution inputs or complex conversational flows.

In contrast, language models can provide highly fluent conversations by framing the input token sequence as a series of interactions between a user and an assistant (see Figure A.1 in the Appendix for details on the prompt used). This is effectively the opposite of a rule-based approach, where all knowledge and state tracking are implicitly tracked through tokens. Crucially, the language model does not explicitly model any step and solely relies on its weights and the context to keep track of the conversational state. Figure 7.2 shows a conversation about a recipe with GPT-2-large, a language model of a comparable size to the NDP. The language model is conversational, yet it easily loses the thread of the conversation when tasked with going back to previously discussed topics.

The challenge lies in bridging the gap between a user's contextual, often underspecified, natural language. GRILLBot's core philosophy leans into these limitations rather than trying to fix them within the language model itself. We externalise knowledge (recipes, facts) and reasoning (task progression) to structured, easily updatable components. We then use pre-trained language models for their primary strength: understanding ambiguous user intent within a rich context. The language model's role is to direct the system's policy by generating code, not to hold the state or knowledge itself.

7.2 The GRILLBot System Architecture

GRILLBot is orchestrated by a central component that manages the conversational flow and invokes other modules. Its architecture is built around the explicit separation of knowledge, tools, and the agentic logic that connects them.



Hi, can you help me with a recipe for creamy zucchini pasta?

Absolutely! Here's a simple recipe for creamy zucchini pasta:
Ingredients:

- 2 medium zucchinis, sliced
- ...
- Fresh basil for garnish

Thanks! Would it work if I use courgettes instead of zucchinis?

Yes, absolutely! Courgettes and zucchinis are actually the same vegetable — "courgette" is the British term, and "zucchini" is the American term. So you can use them interchangeably.

Perfect. Now, a bit later I remembered: you said something about basil, right? Should I add it while cooking or just on top?

Oh yes! Basil is great with pasta. If you're making a tomato-based sauce, you can simmer it with the sauce. Otherwise, fresh basil works well as a garnish on top of the pasta just before serving to preserve its flavor.

Wait, we weren't doing tomato-based sauce, were we? This is the creamy zucchini pasta.

Absolutely! For creamy sauces, I recommend simmering the tomatoes gently to let the flavors develop. Start by sautéing some onions and garlic, then add canned tomatoes and let it cook down before adding your cream. The zucchini can be added at the same time as the tomatoes.

Just go back to the first recipe we were talking about, the one with zucchini as the main ingredient.

Of course! There are many great recipes with zucchini as a main ingredient. Let's go back to the zucchini and basil pesto. [...] Zucchini can be added towards the end to keep it fresh and vibrant.

...okay. Never mind.

Figure 7.2: Conversation about Creamy Zucchini Pasta with GPT-2-large (temperature=0.2). Notice the fluent interactions between the user and the assistant, yet the lack of grounding allows GPT-2 to lose the thread of conversation, resulting in a frustrating user experience.

7.2.1 Data Structures

TaskGraphs

To enable robust, long-horizon task guidance, GRILLBot externalises all task-specific knowledge into a dedicated data structure called the *TaskGraph*, separating it from the language model's parametric memory. The TaskGraph serves as the formal, programmatic representation of a task (e.g., a recipe or DIY project), designed as a directed acyclic graph (DAG) to model the complex flow of actions and information dependencies. It is composed of several heterogeneous node types, each serving a specific role:

- **Steps Node:** Represents a single instruction for the user, containing a short summary, a detailed description, and metadata for multimodal presentation (e.g., images for screen devices).
- **Requirements Node:** Defines the necessary tools and ingredients for the task, typically linked to the specific Steps where they are used.
- **Conditions Node:** Acts as a yes/no gate whose state is resolved dynamically based on user input or system state, allowing the TaskGraph to be adaptive (e.g., "Do you want to add a glaze?").
- **Logic Node:** Represents logical operations (AND, OR, NOT) to create compact dependency structures. An OR (ANY) node can model parallel dependencies, while a NOT node combined with a Condition node creates an IF-ELSE switch.
- **Actions Node:** Represents an action to be performed automatically by the system, such as starting a timer or triggering live retrieval of domain knowledge.
- **Extra Information Node:** Contains supplementary, non-critical content like tips or fun facts to enrich the user experience.

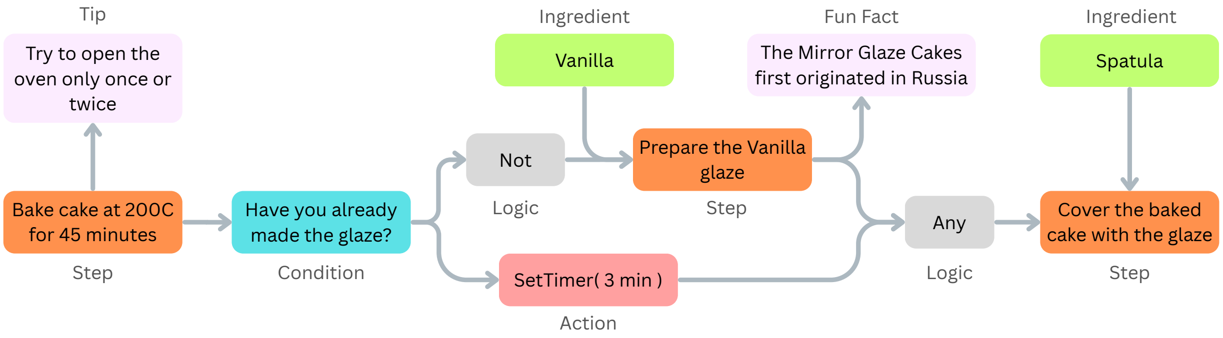


Figure 7.3: Example section of a TaskGraph for baking a cake.

As shown in Figure 7.3, this combination of nodes allows for highly adaptive interactions where GRILLBot can take initiative, such as proactively offering tips or automatically starting a timer. A key practical advantage is that TaskGraphs are manually inspectable and editable, allowing developers to debug flows and optimise tasks without retraining the core AI models. Ultimately, the TaskGraph is the grounded, symbolic data structure that the Neural Decision Parser (the program generator) learns to navigate and manipulate.

Search Indices and Conversational Phases

In addition to TaskGraphs, which are built by crawling and parsing sources like wikiHow and Epicurious, GRILLBot uses search indices for retrieval. The entire user journey is modelled through distinct phases: Domain selection, Planning (task search), Validation (confirming requirements), Execution (step-by-step guidance via the TaskGraph), and Farewell.

7.3 The Tools

In GRILLBot, tools are discrete, programmatic modules that perform specific functions when called by the program generator. Key tools include:

- **Task Navigation:** A tool that interacts with the active TaskGraph to move between nodes (e.g., `next()`, `previous()`, `step_select(4)`).
- **General Question Answering:** A tool for answering out-of-domain or general knowledge questions, which calls an external API (Amazon Evi). It takes a natural language question and returns a factual response, acting as an external data source.
- **Task-Specific Question Answering:** A tool that answers questions about the current task (e.g., "How much oil do I need?") by creating a textual representation of the current TaskGraph state and querying it with a specialised QA model, specifically UnifiedQA [70].
- **Task Search:** A tool that takes a user's natural language query and retrieves the most relevant TaskGraphs from the search indices.

7.4 Agentic Components

As outlined in Chapter 2, there are three components used to represent the agent. Below, we describe these and how they relate to GRILLBot:

- **Task Processor & "Tool Required" Detector:** These two functions are combined into a single, lightweight, embedding-based **Intent Classifier**. It uses fine-tuned BERT embeddings [32] to perform initial triage on the user's utterance, handling default cases (e.g., harmful requests) or deciding which general tool category (e.g., navigation, QA, search) is required. This lightweight design was crucial for meeting the low-latency requirements of the competition.

- **Program Generator:** The NDP serves as the Program Generator. It translates the user's intent and the current system state into a specific, executable command. Its output is a Python function call with appropriate parameters, selecting the precise tool and configuring its execution.

The Python code generated by the NDP is parsed and executed by the Orchestrator. This execution calls the dedicated tool (e.g., the Task Navigation module). The output from the executed tool (e.g., the text for the next step) is then returned to the Orchestrator to formulate a response.

7.5 The Neural Decision Parser

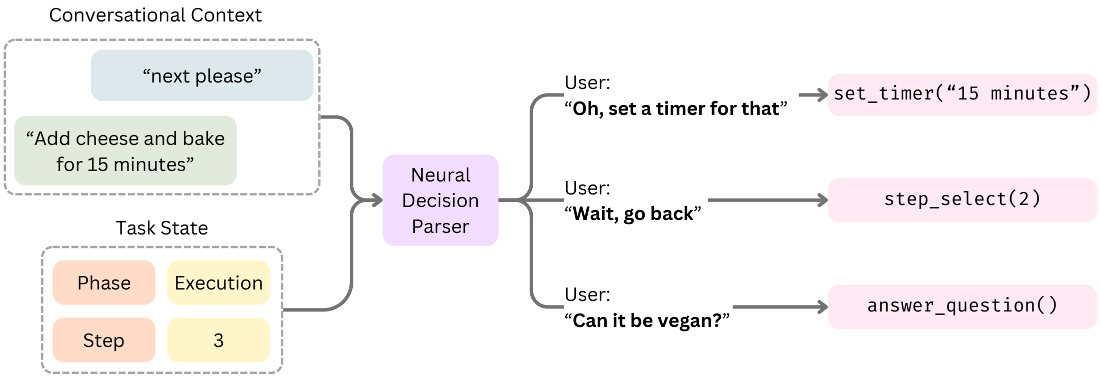


Figure 7.4: Input-output examples of the Neural Decision Parser, showcasing how user utterances and task state are translated into code.

The neural decision parser's (NDP) central role is to generate the system's policy for the next turn as a line of code. This "policy as code" approach differs from traditional hard-coded dialogue policies, as the NDP learns the mapping from conversation state to system action from data.

The NDP was trained on a parallel dataset of (utterance, system state) pairs mapped to target Python code. The system state is a key-value map containing relevant information like the selected recipe, current step, conversation phase, and turn history. This data was created through a combination of human annotation and large-scale synthetic data augmentation using GPT-3, leveraging the principles of weak supervision explored in Chapter 5. Each turn in these conversations was annotated with the appropriate command (the target Python code) that the NDP should predict given the preceding context and the current user utterance. This supervised training dataset comprised approximately 1,200 annotated turns, covering diverse conversational stages and task complexities.

The pre-trained T5-large model was fine-tuned on this dataset. The use of a pre-trained language model was crucial, as it endowed the NDP with strong foundational language understanding capabilities out-of-the-box, including coreference resolution, semantic parsing for parameter extraction (e.g., extracting a timespan for a timer), and implicit state tracking, which were then specialised for the task-oriented dialogue domain.

The output of the NDP is a command in a parametrised Python function. As shown in Figure 7.1, this includes commands for task navigation (`next()`, `step_select(step=Int)`), tool invocation (`answer_question()`, `search(...)`), and utility functions (`timer(span=Stri`
This ensures the language model's output is constrained to a set of valid, executable, and interpretable actions.

```
# User specifies which task to execute
> select(option=Int)

# Catch all for user questions
> answer_question()

# Catch all for task search
# Vague and Theme query categories
> search(vague=Bool, theme=String)

# Go to prior node
> previous()

# Go to next scheduled node
> next()

# Navigate to specific task steps
> step_select(step=Int)

# Set timer with parsed time span
> timer(span=String)

# Provide details about a step
> chit_chat()
```

Listing 7.1: A sample of the Neural Decision Parser output. These Python commands with intent-based functions and parameterised arguments are generated by the T5 model at inference time to direct system behaviour.

7.5.1 Architecture (T5 Encoder-Decoder)

The NDP is architected around the T5-large transformer model [114], leveraging its encoder-decoder architecture compared to decoder-only language models. The input to the NDP is carefully structured to enable effective reasoning:

- **Encoder Input (Conversational State):** The conversation history (previous user and system utterances) and the current task state (e.g., current step, available options) are concatenated and fed into the T5 encoder. The encoder processes this information, creating contextualised embeddings. This can be conceptualised as the encoder building a "soft neural database" of the conversation, where the state is represented as distributed patterns within the model's hidden layers rather than explicit key-value pairs. This encoded state remains static for the current turn.
- **Decoder Input (User Utterance):** The latest user utterance is fed to the T5 decoder.
- **Cross-Attention Mechanism:** During generation, the decoder attends to the encoder's output via cross-attention layers. This allows each token generated by the decoder to be conditioned not only on the current user utterance but critically, on the entire encoded representation of the conversational state.
- **Output (Policy as code):** The decoder auto-regressively generates a sequence of tokens that form an executable Python command. This command represents the decided system policy for the current turn.
- **System execution:** The system parses the produced command and calls any specified procedures: question answering, task advancement, timers, etc. These procedures are shown in Figure 7.1. The result of the system execution is always a system response since it is the primary medium to inform the user of the task state. For instance, when advancing a recipe task, the system might read the next step aloud.

This architecture, depicted in Figure 7.5, is designed to separate the processing of the relatively stable conversational context from the immediate user input, allowing the model to efficiently integrate both for policy generation. For instance, as illustrated in Figure 7.4, if the task state indicates the current step is 3 and the user says "Wait, go back," the NDP is trained to generate code like `step_select(2)`.

7.6 System Operation: The Tool Use Loop

GRILLBot operates as a continuous tool use system, interfacing its neural components with rigid data structures via programs in a turn-by-turn loop.

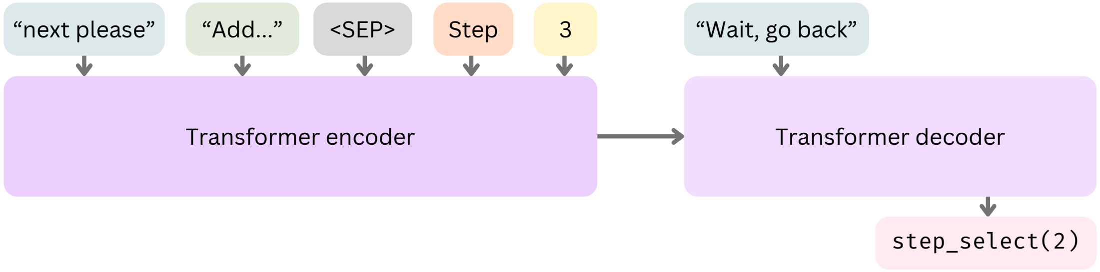


Figure 7.5: Transformer-based architecture of the Neural Decision Parser. Conversational context and task state are encoded, while the user utterance drives the decoder to generate policy code via cross-attention.

7.6.1 State Management

A persistent state dictionary holds the complete state of the conversation. This dictionary represents the data state T_i at the beginning of a turn. After the tool generated by the NDP is executed, the dictionary is updated with the results, representing the new state T_{i+1}. This explicitly offloads state persistence to a deterministic digital system, freeing the language model from this responsibility and preventing the state-loss issues seen in pure language-model-based systems. This separation of concerns allows for independent optimisation: the NDP can be retrained to call new tools without altering the tools themselves, and vice-versa. The Orchestrator uses the final state T_{i+1} to generate the next system utterance to the user, completing the loop.

7.7 Method

GRILLBot is evaluated in two settings: offline and online.

Offline, we curate a development and test set for the NDP from human annotators. Each sample in these sets follows the training format but uses out-of-distribution tasks that are not seen in the training set. We also ensure all user utterances are unique to measure generalisation. For metrics, we use exact match over the function name and the parameters. We then combine the overall scores to give a unified measure of effectiveness using per-sample averaging.

Online, GRILLBot is evaluated in a real-time setting with users across the United States querying multi-modal (screen and voice-only) Amazon Alexa devices live. Users voluntarily initiate a conversation with GRILLBot by navigating internal Alexa menus. During the competition, a random bot out of N participating teams is selected to lead the interaction until completion. At the end of an interaction, when a user completes the task, decides to exit, or times out due to a system malfunction, the user is prompted to rate the conversation from 0 to 5. This rating is subjective to each user, with each very rarely interacting with the system more than once. We estimate the total number of interactions during a single month to exceed 10,000 conversations.

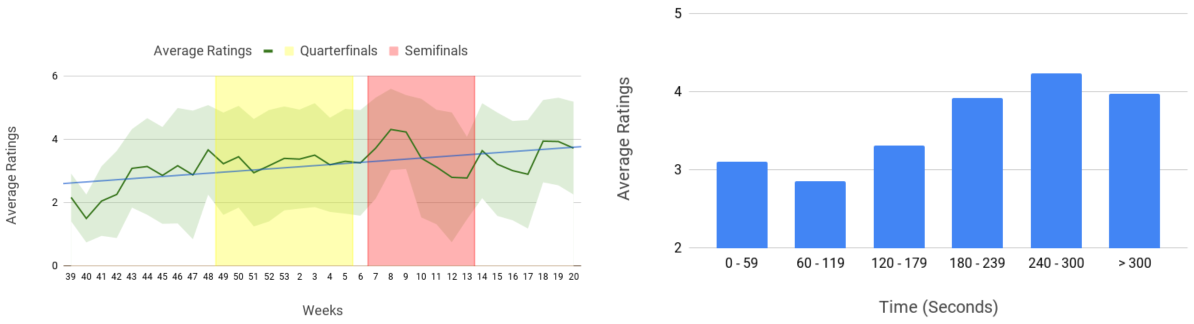
Model	**Development Set Accuracy**			**Test Set Accuracy**		
	Function	**Argument**	**Overall**	**Function**	**Argument**	**Overall**
T5-small	0.605	0.651	0.623	0.601	0.668	0.622
T5-base	0.854	0.735	0.781	0.891	0.799	0.828
T5-large	**0.901**	**0.812**	**0.838**	**0.925**	**0.835**	**0.863**

Table 7.1: Decomposed NDP performance on combined intent and attribute accuracy. "Function" refers to the accuracy of predicting the correct DSL command. "Argument" refers to the exact match accuracy of the parameters, assuming the function was correct. "Overall" is the joint accuracy where both function and all arguments must be correct.

7.8 Results

Table 7.1 shows the exact match results for different sizes of the NDP on the manually curated dev and test sets. The T5-large-based NDP achieved a combined intent and attribute accuracy of 86.3% on the test set, significantly outperforming smaller T5-base (82.8%) and T5-small (62.2%) models, demonstrating the benefits of scale for this task while still being small enough for real-time interaction. We find, under closer inspection, that the NDP is very proficient in common function calls like task navigation like "next step please" yet struggles with more complex timer-based interactions. We find the performance of the larger model sufficient for wider deployment and use it in the live user setting. We also observe that the code produced is almost always well-formatted, with 98% of generated code parsing correctly, indicating that the transformer-based NDP is able to implicitly model syntax, further supporting our claims from Chapters 4 & 5. Throughout the course of the 9-month-long competition, we updated the datasets used to train the NDP several times. These updates consisted of identifying low-performance function calls and adding new samples to the train, dev, and test sets. Figure 7.6a helps put these NDP updates in context, showing the public perception of the system over time, with week 53 indicating the last week of 2021 and week 2 indicating the second week of 2022. Four major releases to the NDP were made throughout the competition: at week 40, week 49, week 6, and week 17. At weeks 4, 6, and 17, we observe a noticeable improvement in ratings relative to the weeks prior. The first update is particularly influential since we observed that navigational function calls were crucial for baseline user experience and found initial performance unsatisfactory, with user ratings reaching less than 2 out of 5 stars. It is important to note that while the NDP updates correlate well with overall GRILLBot performance, these scores are influenced by many factors, such as improved TaskGraph corpora and other system improvements.

Overall, user ratings for GRILLBot showed a consistent upward trend throughout the competition, peaking at an average of just over 4 out of 5 stars (Figure 7.6a). This indicates that the tool use architecture led to an increasingly robust and satisfying user experience. Furthermore, analysis showed that ratings were positively correlated with conversation length, peaking for conversations lasting 4-5 minutes (Figure 7.6b). This demonstrates the system's ability to



(a) Average ratings throughout the competition.

(b) Average ratings by conversation length.

Figure 7.6: User rating analysis for GRILLBot during the competition.

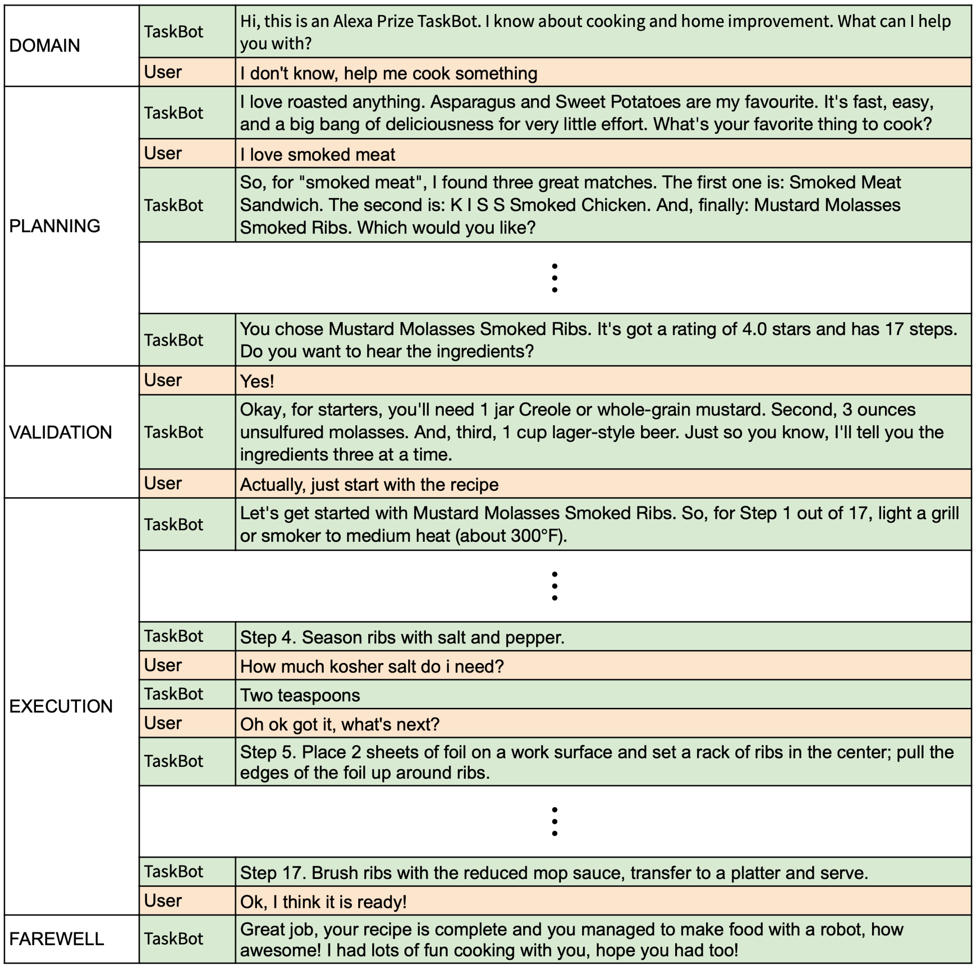
maintain state and coherence over long interactions, a direct benefit of its externalised state management and tool-centric design. Figure 7.7 depicts a 7-minute example conversation with a team member.

7.9 Conclusion

GRILLBot serves as a real-world validation of the thesis's central claim: augmenting language models with the ability to generate and use programmatic tools is a highly effective paradigm. The Neural Decision Parser successfully demonstrates how a language model can function as a flexible and context-aware policy engine, translating unconstrained language into precise, executable actions. Within the context of the framework described in Chapter 2, the neural decision parser acts as an effective program generator. It supports Hypothesis 1.1 by demonstrating that transformers can generate well-formed code from natural language. More broadly, GRILLBot and its numerous iterations throughout the competition show that programs are a viable and flexible option for interfacing between neural and digital components of a system. This hybrid approach effectively combines the pattern-recognition strengths of language models with the reliability and scalability of structured data and programmatic execution. The first-place victory in the Amazon Alexa Prize TaskBot competition provides strong external validation that this tool-augmented architecture leads to state-of-the-art performance in complex, task-oriented conversational AI.

7.10 Broader Context and Subsequent Developments

The approach the NDP takes was not an established design pattern in 2021. OpenAI's function calling API, released in 2023, formalised the same approach at the infrastructure level. The interface it defines between a language model and executable actions is structurally identical to what the NDP implements: the model selects a named function and populates its arguments,



DOMAIN	TaskBot	Hi, this is an Alexa Prize TaskBot. I know about cooking and home improvement. What can I help you with?
	User	I don't know, help me cook something
PLANNING	TaskBot	I love roasted anything. Asparagus and Sweet Potatoes are my favourite. It's fast, easy, and a big bang of deliciousness for very little effort. What's your favorite thing to cook?
	User	I love smoked meat
	TaskBot	So, for "smoked meat", I found three great matches. The first one is: Smoked Meat Sandwich. The second is: K I S S Smoked Chicken. And, finally: Mustard Molasses Smoked Ribs. Which would you like?
		⋮
	TaskBot	You chose Mustard Molasses Smoked Ribs. It's got a rating of 4.0 stars and has 17 steps. Do you want to hear the ingredients?
VALIDATION	User	Yes!
	TaskBot	Okay, for starters, you'll need 1 jar Creole or whole-grain mustard. Second, 3 ounces unsulfured molasses. And, third, 1 cup lager-style beer. Just so you know, I'll tell you the ingredients three at a time.
	User	Actually, just start with the recipe
EXECUTION	TaskBot	Let's get started with Mustard Molasses Smoked Ribs. So, for Step 1 out of 17, light a grill or smoker to medium heat (about 300°F).
		⋮
	TaskBot	Step 4. Season ribs with salt and pepper.
	User	How much kosher salt do i need?
	TaskBot	Two teaspoons
	User	Oh ok got it, what's next?
	TaskBot	Step 5. Place 2 sheets of foil on a work surface and set a rack of ribs in the center; pull the edges of the foil up around ribs.
		⋮
	TaskBot	Step 17. Brush ribs with the reduced mop sauce, transfer to a platter and serve.
	User	Ok, I think it is ready!
FAREWELL	TaskBot	Great job, your recipe is complete and you managed to make food with a robot, how awesome! I had lots of fun cooking with you, hope you had too!

Figure 7.7: Example conversation with phases for each utterance. The user utterances are from a team member in the interest of preserving user privacy.

with downstream components responsible for execution.

The multi-turn structure of GRILLBot's tool use loop, in which executed code output is re-injected into the following turn's context, has since become the defining pattern of LLM-based agents. Frameworks such as LangChain, LlamaIndex, and the agent loop formalised in OpenAI's Assistants API all implement variants of this observe-act-observe cycle. The Model Context Protocol [3] standardises how agents discover and invoke tools across arbitrary backends, embedding this loop at the protocol level. The specific state management problem GRILLBot addressed to maintain coherent task context across the turns of a recipe-oriented conversation hinted at the methods used to handle longer-horizon tasks these agentic systems in 2026.

Chapter 8

Conclusion

This thesis has explored augmenting language models by enabling them to write and execute computer programs as tools. We have demonstrated that by equipping language models with the ability to generate and strategically utilise programs, we can significantly enhance their capacity to reason about and interact with structured information. This approach leverages the distinct strengths of probabilistic language models and deterministic program execution.

Our contributions fall into two main categories: first, improving the foundational capability of language models to generate code from natural language (Chapters 4 & 5), and second, demonstrating how this code-generation ability can be applied to enhance performance on tasks involving structured data, particularly in tabular question answering (Chapter 6) and conversational agents (Chapter 7). The research presented here makes several key contributions that validate both components of this thesis statement.

First, regarding language models' ability to generate code, I demonstrated in Chapter 4 that transformer architectures combined with retrieval techniques significantly enhance natural language to code translation performance. This validates Hypothesis 1, showing the superiority of this approach over traditional LSTM-based methods [47]. Building on this foundation, Chapter 5 addressed Hypothesis 2 by revealing that weak supervision signals from pre-trained language models, particularly through infilling objectives, can further improve translation accuracy. The research showed that jointly pre-training on aligned code and natural language yields optimal results for translation similarity and parsing accuracy.

Second, concerning language models' ability to use code as tools, I validated Hypothesis 3 through several key findings. In Chapter 6, I demonstrated that language models can effectively generate question-specific programs to filter tabular data, thereby improving the accuracy of language-model-based TQA systems. ToolWriter successfully addresses challenges posed by large table sizes and complex queries [44], demonstrating that language models can leverage generated programs as tools to enhance their performance on structured data tasks.

The practical implications of these findings extend beyond theoretical validation. As demonstrated in Chapter 7, the Neural Decision Parser I developed for the GRILLBot system [45, 46] successfully applied these principles in a real-world setting. The system's victory in the 2021-2022 Amazon Alexa Prize TaskBot competition provides compelling evidence of the practical viability of language models generating and utilising programs in complex conversational AI scenarios.

Through systematic investigation of the hypotheses outlined in Section 1.6, this thesis has established that language models can not only generate programs effectively but also strategically employ them as tools to enhance their performance on structured information tasks. This capability represents a significant advancement in the field of natural language processing, opening new avenues for improving AI systems' ability to reason about and interact with structured data.

8.1 Future Directions

The techniques developed in this thesis, natural language to code translation and the use of generated programs as an interface to external tools, sit at the foundation of a broader shift toward agentic language model systems. Continuing in this style of work with modern language models and techniques suggests two particularly promising directions:

1. **Modelling uncertainty and human-in-the-loop clarification:** A persistent challenge for language models used as agents is that they struggle to know what they do not know, producing confidently wrong outputs in the face of ambiguous or underspecified inputs. Early progress on this problem has been demonstrated by training models to express calibrated uncertainty in natural language [85]. Extending this capability to agentic tool-use settings, where a model must decide when to act, when to ask for clarification, and when to defer, becomes increasingly important as autonomous loops extend over longer horizons, as seen in contemporary autonomous coding agents.

2. **Scaling and verifying longer autonomous loops:** As tool-use trajectories lengthen, stability degrades: errors compound and become difficult to detect or attribute. A natural research arc follows the progression of available supervision signals, from supervised labels of the kind used in ToolWriter, to ground-truth-verifiable rewards on tasks with definite answers, to reward models trained on human preferences, and ultimately to signals extracted from human intervention points or post-trajectory feedback. The central challenge is building robust loops using cheap, accessible forms of supervision before committing to fully autonomous deployment.

Together, these directions reflect the broader evolution of language model research from narrow code generation toward reliable, long-horizon agentic systems, a trajectory to which the work in this thesis provides early foundational steps.

Appendix A

GPT-2 conversational agent

```
You are a helpful cooking assistant. You assist a user by providing
   them with recipe suggestions, ingredients, instructions and
   answering their questions in context. Help the user by walking
   them through any recipe step-by-step.

Format the conversation using <user> and <assistant> tags.

Begin:

<user> Hi, can you help me with a recipe for creamy zucchini pasta?
<assistant> Absolutely! Here's a simple recipe for creamy zucchini
   pasta:

Ingredients:
- 2 medium zucchinis, sliced
- 2 cloves garlic, minced
- 1 tablespoon olive oil
- 1/2 cup heavy cream
- 1/4 cup grated Parmesan cheese
- 200g pasta of your choice
- Salt and pepper to taste
- Fresh basil for garnish

<user> Thanks! Would it work if I use courgettes instead of
   zucchinis?
<assistant> Yes...
```

Listing A.1: GPT-2-large prompt used to assess baseline language model performance on long context cooking tasks.

This figure displays the full prompt used to elicit a conversational, task-oriented cooking assistant persona from the GPT-2-large language model for the baseline demonstration in Chapter 7. It utilises a few-shot learning approach, providing a high-level instruction that defines the assistant's role and capabilities, instructions for formatting the dialogue, and a concrete example of a user-assistant interaction to guide the model's generation. This prompt structure is designed to produce the fluent but ungrounded conversational behaviour seen in Figure 7.2, highlighting the limitations of pure language models that the tool-augmented architecture of GRILLBot aims to overcome.

Appendix B

ToolWriter Detailed Analysis

The main body of Chapter 6 established that the performance of language-model-based Tabular Question Answering (TQA) systems degrades with increasing table size and that targeted, programmatic row filtering can mitigate this issue. This section provides a more granular analysis of the baseline TQA models' performance, breaking down the WikiTableQuestions (WTQ) dev set into functionally distinct subsets. The goal of this analysis is to identify the specific types of reasoning where neural models falter, thereby providing a more detailed justification for the programmatic tool-based approach of ToolWriter.

The following table, Table B.1, presents the exact match (EM) scores for four baseline models—UnifiedSKG, OmniTab, GPT-3, and FlanT5—across various challenging subsets of the WTQ dev set. The subsets are defined by structural or semantic properties of the questions and their corresponding SQL-like annotations from the SQUALL dataset [125]. The cell colouring provides a visual guide, with green indicating stronger performance and red indicating weaker performance relative to the model's average.

Analysis of Data Subsets

The SQUALL annotated subset represents the majority (77.11%) of the data, comprising questions that can be mapped to a formal SQL-like representation. This is the "comfort zone" for TQA models, especially those trained on SQL-related corpora like OmniTab. Conversely, the Non-SQUALL-annotated subset contains questions with more complex, implicit, or non-standard reasoning that do not easily map to SQL. As expected, all models perform significantly worse on this subset, with OmniTab's EM dropping by over 26 points. This stark contrast demonstrates a fundamental weakness: when queries deviate from simple, formalisable structures, the performance of these neural systems degrades sharply. This finding motivates the need for more flexible reasoning mechanisms, such as the executable Python programs generated by ToolWriter.

`+ 1` as offset: This small subset (2.3%) tests simple arithmetic, like finding an adjacent

Data Subset	**UnifiedSKG**	**OmniTab**	**GPT-3**	**FlanT5**	**dataset %**
SQUALL annotated	54.56	66.65	38.62	17.36	77.11
Non-SQUALL-annotated	33.49	40.43	30.4	11.27	22.89
+ 1 as offset	69.23	73.85	47.69	16.92	2.3
requires counting rows	43.14	66.11	27.17	6.72	25.22
count all rows	39.62	66.67	16.98	3.77	5.62
big sub or add	6.58	11.84	55.26	2.63	2.68
1 `where` clause	55.65	70.75	39.92	17.63	44.68
2 `where` clauses	49.42	59.83	41.62	15.32	12.22
count and where	54.41	67.8	39.73	16.35	63.3
answer not in prompt	30.23	31.4	26.74	3.88	9.11
multiple answers	41.94	51.61	32.26	0	2.19
one select	55.92	68.33	38.15	17.88	63.79
one select and one where	55.99	71.2	40.06	17.7	43.91
has duplicate columns	30.59	41.18	21.18	10.59	3

Table B.1: Exact match performance of systems according to subsets of the WikiTableQuestions dev set.

rank or value. The top models (UnifiedSKG, OmniTab) perform very well here, suggesting they have learned to handle trivial, single-step table lookups.

big sub or add: This subset requires arithmetic with larger numbers. Performance for all models except GPT-3 collapses catastrophically. UnifiedSKG and OmniTab, despite their specialised training, score only 6.58% and 11.84% respectively. GPT-3's comparatively strong performance (55.26%) is likely due to its large parameter count yet falls short of much higher accuracies proportional to the size difference. This provides clear evidence for Hypothesis 3: LLMs are unreliable for precise calculation at scale, a task perfectly suited for a deterministic program.

Counting is an algorithmic procedure that LLMs will approximate rather than execute.

- **requires counting rows:** On this large subset (25.22%), most models struggle significantly, with scores for UnifiedSKG, GPT-3, and FlanT5 dropping well below their baseline. OmniTab maintains high performance, likely due to its extensive pre-training on SQL-like tasks, which frequently involve `COUNT(*)` operations.

- **count all rows:** A simple query asking for the total number of rows. The poor performance of most models here reinforces the findings from Chapter 6 that even simple tasks become difficult when the input table is large and requires processing all rows.

These results further validate the core premise of ToolWriter. While an LLM struggles to count rows by processing a linearised string, a program can do so trivially and accurately with a command like `len(table)`.

Compositional Reasoning (`where` clauses) This analysis examines how models handle increasing logical complexity.

- **1 `where` clause:** Questions requiring a single filter condition are handled relatively well, with performance close to the overall SQUALL baseline.
- **2 `where` clauses:** Adding a second filter condition causes a noticeable drop in performance for all models. This demonstrates the difficulty LLMs face with compositional generalisation. Each additional logical step increases the probability of error.

This degradation supports the thesis that procedural tasks are better handled by programs, which are inherently compositional. ToolWriter's approach of generating a single program with multiple filter conditions (`lambda row: condition1 and condition2`) directly addresses this weakness by offloading the compositional logic to a reliable execution environment.

The **answer not in prompt** subset contains questions where the answer must be computed or inferred rather than simply extracted from a cell (e.g., from an arithmetic operation). As seen with the "big sub or add" subset, performance plummets. This is a critical failure mode, as it shows the models' over-reliance on extractive, pattern-matching strategies.

The **has duplicate columns** subset, though small, reveals that ambiguous table schemas are highly detrimental to all models. Their performance drops significantly, indicating they lack robust mechanisms for disambiguating column references.

Overall Implications

This detailed analysis provides strong, quantitative evidence for the claims made in Chapter 6 and the overarching thesis. The baseline neural models, while capable on simple, extractive queries, exhibit systemic weaknesses in several key areas:

1. **Numerical Reasoning:** They are unreliable for precise counting and arithmetic, especially with larger numbers.
2. **Compositional Logic:** Their performance degrades as the number of logical steps (e.g., filter conditions) increases.
3. **Scalability:** Their effectiveness diminishes on simple tasks when the input data (table size) grows.
4. **Non-Extractive Tasks:** They struggle when the answer must be computed rather than found verbatim in the table.

These identified weaknesses are precisely the strengths of programmatic execution. A computer program is deterministic, perfectly reliable for arithmetic, inherently compositional, and scalable. Therefore, this analysis validates the architectural choice of ToolWriter: using a language model for what it does best (understanding the natural language query and schema) while offloading the procedural, logical, and numerical reasoning to a generated program. This hybrid approach directly targets the failure modes of pure neural systems, providing a robust path towards more capable and reliable reasoning over structured data.

Bibliography

[1] Acharya, A., Adhikari, S., Agarwal, S., Auvray, V., Belgamwar, N., Biswas, A., Chandra, S., Chung, T., Fazel-Zarandi, M., Gabriel, R., Gao, S., Goel, R., Hakkani-Tür, D.Z., Jezabek, J., Jha, A., Kao, J.Y., Krishnan, P., Ku, P., Goyal, A., Lin, C.W., Liu, Q., Mandal, A., Metallinou, A., Naik, V.I., Pan, Y., Paul, S., Perera, V., Sethi, A., Shen, M., Strom, N., Wang, E.: Alexa conversations: An extensible data-driven approach for building task-oriented dialogue systems. In: Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Demonstrations. pp. 125–132 (2021). https://doi.org/10.18653/v1/2021.naacl-demos.15, `https://aclanthology.org/2021.naacl-demos.15/`

[2] Allamanis, M., Barr, E.T., Devanbu, P.T., Sutton, C.: A survey of machine learning for big code and naturalness. ACM Computing Surveys **51**(4), 1–37 (2018). https://doi.org/10.1145/3212695, `https://dl.acm.org/doi/10.1145/3212695`

[3] Anthropic: Introducing the model context protocol (2024), `https://www.anthropic.com/news/model-context-protocol`

[4] Bahdanau, D., Cho, K., Bengio, Y.: Neural machine translation by jointly learning to align and translate. In: International Conference on Learning Representations (ICLR) (2015). https://doi.org/10.48550/arXiv.1409.0473, `https://arxiv.org/abs/1409.0473`

[5] Baltes, S., Diehl, S.: Towards a theory of software development expertise. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 187–200. ACM (2018). https://doi.org/10.1145/3236024.3236061, `https://dl.acm.org/doi/10.1145/3236024.3236061`

[6] Bengio, Y., Ducharme, R., Vincent, P., Jauvin, C.: A neural probabilistic language model. Journal of Machine Learning Research **3**, 1137–1155 (2003). https://doi.org/10.5555/944919.944966, `https://dl.acm.org/doi/10.5555/944919.944966`

[7] Berant, J., Chou, A.K., Frostig, R., Liang, P.: Semantic parsing on freebase from question-answer pairs. In: Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing. pp. 1533–1544 (2013). https://doi.org/10.18653/v1/D13-1160, `https://aclanthology.org/D13-1160/`

[8] Berard, A., Calapodescu, I., Roux, C.: Naver labs europe's systems for the wmt19 machine translation robustness task. In: Proceedings of the Fourth Conference on Machine Translation (Volume 2: Shared Task Papers, Day 1). pp. 526–532 (2019). https://doi.org/10.18653/v1/W19-5361, `https://aclanthology.org/W19-5361/`

[9] Bhattamishra, S., Ahuja, K., Goyal, N.: On the ability and limitations of transformers to recognize formal languages. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP). pp. 7096–7116. Association for Computational Linguistics, Online (2020). https://doi.org/10.18653/v1/2020.emnlp-main.576, `https://aclanthology.org/2020.emnlp-main.576/`

[10] Bobrow, D.G.: Gus, a frame driven dialog system. Artificial Intelligence **8**(2), 155–173 (1977). https://doi.org/10.1016/0004-3702(77)90018-2, `https://doi.org/10.1016/0004-3702(77)90018-2`

[11] Bocklisch, T., Faulkner, J., Pawlowski, N., Nichol, A.: Rasa: Open source language understanding and dialogue management. ArXiv **abs/1712.05181** (2017). https://doi.org/10.48550/arXiv.1712.05181, `https://arxiv.org/abs/1712.05181`

[12] Bohus, D., Rudnicky, A.I.: The RavenClaw dialog management framework: Architecture and systems. Computer Speech and Language **23**(3), 332–361 (2009). https://doi.org/10.1016/j.csl.2008.10.001, `https://doi.org/10.1016/j.csl.2008.10.001`

[13] Brey, P.: The epistemology and ontology of human-computer interaction. Minds and Machines **15**, 383–398 (2005). https://doi.org/10.1007/s11023-005-9003-1, `https://doi.org/10.1007/s11023-005-9003-1`

[14] Bril, B., Smaers, J.B., Steele, J., Rein, R., Nonaka, T., Dietrich, G., Biryukova, E.V., Hirata, S., Roux, V.: Functional mastery of percussive technology in nut-cracking and stone-flaking actions: experimental comparison and implications for the evolution of the human brain. Philosophical Transactions of the Royal Society B: Biological Sciences **367**(1585), 59–74 (2012). https://doi.org/10.1098/rstb.2011.0147, `https://royalsocietypublishing.org/doi/10.1098/rstb.2011.0147`

[15] Bromley, J., Bentz, J.W., Bottou, L., Guyon, I.M., LeCun, Y., Moore, C., Säckinger, E., Shah, R.: Signature verification using a "siamese" time delay neural network. Int. J. Pattern Recognit. Artif. Intell. **7**, 669–688 (1993). https://doi.org/10.1142/S0218001493000339, `https://doi.org/10.1142/S0218001493000339`

[16] Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T.J., Child, R., Ramesh, A., Ziegler, D.M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., teusz Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., Amodei, D.: Language models are few-shot learners. In: Advances in Neural Information Processing Systems. vol. 33 (2020). https://doi.org/10.48550/arXiv.2005.14165, `https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html`

[17] Bueno, M.C., Gemmell, C., Dalton, J.S., de Alencar Lotufo, R., Nogueira, R.: Induced natural language rationales and interleaved markup tokens enable extrapolation in large language models. In: Proceedings of the 1st Workshop on Mathematical Natural Language Processing (MathNLP 2022). pp. 22–31. Association for Computational Linguistics, Abu Dhabi, UAE (2022). https://doi.org/10.18653/v1/2022.mathnlp-1.3, `https://aclanthology.org/2022.mathnlp-1.3/`

[18] Cao, N.D., Aziz, W., Titov, I.: Editing factual knowledge in language models. In: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing. pp. 6491–6506 (2021). https://doi.org/10.18653/v1/2021.emnlp-main.522, `https://aclanthology.org/2021.emnlp-main.522/`

[19] Carroll, J.M. (ed.): Designing Interaction: Psychology at the Human-Computer Interface. Cambridge University Press (1991), `https://www.cambridge.org/us/universitypress/subjects/computer-science/computing-and-society/designing-interaction-psychology-human-computer-interface`

[20] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pondé, H., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F.P., Cummings, D.W., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W.H., Nichol, A., Babuschkin, I., Balaji, S., Jain, S., Carr, A., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M.M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S.,

Sutskever, I., Zaremba, W.: Evaluating large language models trained on code. arXiv preprint **abs/2107.03374** (2021). https://doi.org/10.48550/arXiv.2107.03374, `https://arxiv.org/abs/2107.03374`, no peer-reviewed venue; arXiv only

[21] Chen, W.: Large language models are few(1)-shot table reasoners. In: Findings of the Association for Computational Linguistics: EACL 2023. Association for Computational Linguistics, Dubrovnik, Croatia (2023). https://doi.org/10.18653/v1/2023.findings-eacl.83, `https://aclanthology.org/2023.findings-eacl.83/`

[22] Chen, X., Liu, C., Song, D.X.: Tree-to-tree neural networks for program translation. In: Advances in Neural Information Processing Systems (NeurIPS). vol. 31 (2018). https://doi.org/10.48550/arXiv.1802.03691, `https://proceedings.neurips.cc/paper_files/paper/2018/hash/d759175de8ea5b1d9a2660e45554894f-Abstract.html`

[23] Cheng, Z., Xie, T., Shi, P., Li, C., Nadkarni, R., Hu, Y., Xiong, C., Radev, D.R., Ostendorf, M., Zettlemoyer, L.S., Smith, N.A., Yu, T.: Binding language models in symbolic languages. In: International Conference on Learning Representations (ICLR) (2023). https://doi.org/10.48550/arXiv.2210.02875, `https://openreview.net/forum?id=lH1PV42cbF`

[24] Cho, K., van Merrienboer, B., Çaglar Gülçehre, Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y.: Learning phrase representations using RNN encoder–decoder for statistical machine translation. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP). pp. 1724–1734. Association for Computational Linguistics, Doha, Qatar (2014). https://doi.org/10.3115/v1/D14-1179, `https://aclanthology.org/D14-1179/`

[25] Chung, H.W., Hou, L., Longpre, S., Zoph, B., Tay, Y., Fedus, W., Li, E., Wang, X., Dehghani, M., Brahma, S., Webson, A., Gu, S.S., Dai, Z., Suzgun, M., Chen, X., Chowdhery, A., Valter, D., Narang, S., Mishra, G., Yu, A.W., Zhao, V., Huang, Y., Dai, A.M., Yu, H., Petrov, S., Chi, E.H., Dean, J., Devlin, J., Roberts, A., Zhou, D., Le, Q.V., Wei, J.: Scaling instruction-finetuned language models. Journal of Machine Learning Research **25**(70), 1–53 (2024). https://doi.org/10.48550/arXiv.2210.11416, `https://jmlr.org/papers/v25/23-0870.html`

[26] Churchill, S.E.: Hand morphology, manipulation, and tool use in Neandertals and early modern humans of the Near East. Proceedings of the National Academy of Sciences **98**(6), 2953–2955 (2001). https://doi.org/10.1073/pnas.061032198, `https://www.pnas.org/doi/full/10.1073/pnas.061032198`

[27] Cohen, R., Biran, E., Yoran, O., Globerson, A., Geva, M.: Evaluating the ripple effects of knowledge editing in language models. Transactions of the Association for Computational Linguistics **12**, 283–298 (2023). https://doi.org/10.1162/tacl_a_00644, `https://aclanthology.org/2024.tacl-1.16/`

[28] Crichton, W., Krishnamurthi, S.: Profiling programming language learning. Proceedings of the ACM on Programming Languages **8**(OOPSLA1), 29–54 (2024). https://doi.org/10.1145/3649812, `https://dl.acm.org/doi/10.1145/3649812`

[29] Croft, W.B., Metzler, D., Strohman, T.: Search Engines: Information Retrieval in Practice. Pearson Education (2009)

[30] Cronen-Townsend, S., Zhou, Y., Croft, W.B.: Predicting query performance. In: Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 299–306 (2002), `https://dl.acm.org/doi/10.1145/564376.564428`

[31] DeepSeek-AI: DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. ArXiv **abs/2501.12948** (2025). https://doi.org/10.48550/arXiv.2501.12948, `https://arxiv.org/abs/2501.12948`

[32] Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers). pp. 4171–4186. Association for Computational Linguistics, Minneapolis, Minnesota (2019). https://doi.org/10.18653/v1/N19-1423, `https://aclanthology.org/N19-1423/`

[33] Dong, L., Lapata, M.: Language to logical form with neural attention. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 33–43 (2016). https://doi.org/10.18653/v1/P16-1004, `https://aclanthology.org/P16-1004/`

[34] Dong, L., Lapata, M.: Coarse-to-fine decoding for neural semantic parsing. In: Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 731–742 (2018). https://doi.org/10.18653/v1/P18-1068, `https://aclanthology.org/P18-1068/`

[35] Fasoli, M.: Substitutive, complementary and constitutive cognitive artifacts: Developing an interaction-centered approach. Review of Philosophy and Psychology

9, 671–687 (2017). https://doi.org/10.1007/s13164-017-0363-2, `https://link.springer.com/article/10.1007/s13164-017-0363-2`

[36] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M.: CodeBERT: A pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics: EMNLP 2020. pp. 1536–1547. Association for Computational Linguistics, Online (2020). https://doi.org/10.18653/v1/2020.findings-emnlp.139, `https://aclanthology.org/2020.findings-emnlp.139/`

[37] Ferguson, G., Allen, J.F.: Trips: An integrated intelligent problem-solving assistant. In: Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98) (1998), `https://aaai.org/papers/00567-aaai98-080-trips-an-integrated-intelligent-problem-solving-`

[38] Fodor, J.A., Pylyshyn, Z.W.: Connectionism and cognitive architecture: A critical analysis. Cognition **28**, 3–71 (1988). https://doi.org/10.1016/0010-0277(88)90031-5, `https://doi.org/10.1016/0010-0277(88)90031-5`

[39] Fomicheva, M., Sun, S., Yankovskaya, L., Blain, F., Guzmán, F., Fishel, M., Aletras, N., Chaudhary, V., Specia, L.: Unsupervised quality estimation for neural machine translation. Transactions of the Association for Computational Linguistics **8**, 539–555 (2020). https://doi.org/10.1162/tacl_a_00330, `https://aclanthology.org/2020.tacl-1.35/`

[40] Furrer, D., van Zee, M., Scales, N., Schärli, N.: Compositional generalization in semantic parsing: Pre-training vs. specialized architectures (2020). https://doi.org/10.48550/arXiv.2007.08970, `https://arxiv.org/abs/2007.08970`

[41] Gage, P.: A new algorithm for data compression. The C Users Journal **12**(2), 23–38 (1994), `https://jacobfilipp.com/DrDobbs/articles/CUJ/1994/9402/gage/gage.htm`

[42] Gao, L., Madaan, A., Zhou, S., Alon, U., Liu, P., Yang, Y., Callan, J., Neubig, G.: Pal: Program-aided language models. In: Proceedings of the 40th International Conference on Machine Learning (ICML 2023). Proceedings of Machine Learning Research, vol. 202, pp. 10764–10799. PMLR (2023). https://doi.org/10.5555/3618408.3618843, `https://proceedings.mlr.press/v202/gao23f.html`

[43] Gemmell, C., Dalton, J.: Glasgow representation and information learning lab (grill) at the conversational assistance track 2020. In: Proceedings of the Twenty-Ninth Text

REtrieval Conference (TREC 2020) (2020), `https://trec.nist.gov/pubs/trec29/trec2020.html`

[44] Gemmell, C., Dalton, J.S.: Toolwriter: Question specific tool synthesis for tabular data. In: Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing. pp. 16137–16148 (2023). https://doi.org/10.18653/v1/2023.emnlp-main.1003, `https://aclanthology.org/2023.emnlp-main.1003/`

[45] Gemmell, C., Mackie, I., Dalton, J.: Grillbot: A flexible conversational agent for solving complex real-world tasks. In: Proceedings of the 23rd Annual Meeting of the Special Interest Group on Discourse and Dialogue (SIGDIAL 2022). pp. 621–631 (2022). https://doi.org/10.18653/v1/2022.sigdial-1.63, `https://aclanthology.org/2022.sigdial-1.63/`

[46] Gemmell, C., Mackie, I., Owoicho, P., Rossetto, F., Fischer, S., Dalton, J.: Grillbot: A multi-modal conversational agent for complex real-world tasks. ArXiv **abs/2208.14884** (2022). https://doi.org/10.48550/arXiv.2208.14884, `https://arxiv.org/abs/2208.14884`

[47] Gemmell, C., Rossetto, F., Dalton, J.: Relevance transformer: Generating concise code snippets with relevance feedback. In: Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval (2020). https://doi.org/10.1145/3397271.3401215, `https://dl.acm.org/doi/10.1145/3397271.3401215`

[48] Goldberg, Y.: A primer on neural network models for natural language processing. Journal of Artificial Intelligence Research **57**, 345–420 (2016). https://doi.org/10.1613/jair.4992, `https://jair.org/index.php/jair/article/view/11030`

[49] Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016), `http://www.deeplearningbook.org`

[50] Graves, A., Wayne, G., Danihelka, I.: Neural turing machines (2014). https://doi.org/10.48550/arXiv.1410.5401, `https://arxiv.org/abs/1410.5401`

[51] Gu, X., Zhang, H., Kim, S.: Deep code search. In: Proceedings of the 40th International Conference on Software Engineering (ICSE). pp. 933–944 (2018). https://doi.org/10.1145/3180155.3180167, `https://dl.acm.org/doi/10.1145/3180155.3180167`

[52] Hashimoto, T.B., Guu, K., Oren, Y., Liang, P.S.: A retrieve-and-edit framework for predicting structured outputs. In: Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 31 (NeurIPS 2018). vol. 31. Curran Associates, Inc. (2018). https://doi.org/10.48550/arXiv.1812.01194, `https://proceedings.neurips.cc/paper_files/paper/2018/hash/cd17d3ce3b64f227987cd92cd701cc58-Abstract.html`

[53] Hayati, S.A., Olivier, R., Avvaru, P., Yin, P., Tomasic, A., Neubig, G.: Retrieval-based neural code generation. In: Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing. pp. 925–930 (2018). https://doi.org/10.18653/v1/D18-1111, `https://aclanthology.org/D18-1111/`

[54] Heck, M., van Niekerk, C., Lubis, N., Geishauser, C., Lin, H.C., Moresi, M., Gavsi'c, M.: Trippy: A triple copy strategy for value independent neural dialog state tracking. In: Proceedings of the 21st Annual Meeting of the Special Interest Group on Discourse and Dialogue. pp. 35–44. Association for Computational Linguistics (2020). https://doi.org/10.18653/v1/2020.sigdial-1.4, `https://aclanthology.org/2020.sigdial-1.4/`

[55] Heersmink, R.: Human uniqueness in using tools and artifacts: flexibility, variety, complexity. Synthese **200**(442) (2022). https://doi.org/10.1007/s11229-022-03892-8, `https://link.springer.com/article/10.1007/s11229-022-03892-8`

[56] Herzig, J., Berant, J.: Span-based semantic parsing for compositional generalization. In: Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers). pp. 902–914 (2021). https://doi.org/10.18653/v1/2021.acl-long.74, `https://aclanthology.org/2021.acl-long.74/`

[57] Herzig, J., Nowak, P.K., Müller, T., Piccinno, F., Eisenschlos, J.M.: Tapas: Weakly supervised table parsing via pre-training. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. pp. 4320–4333 (2020). https://doi.org/10.18653/v1/2020.acl-main.398, `https://aclanthology.org/2020.acl-main.398/`

[58] Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Computation **9**, 1735–1780 (1997). https://doi.org/10.1162/neco.1997.9.8.1735, `https://doi.org/10.1162/neco.1997.9.8.1735`

[59] Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., de Las Casas, D., Hendricks, L.A., Welbl, J., Clark, A., Hennigan, T., Noland, E., Millican, K., van den Driessche, G., Damoc, B., Guy, A., Osindero, S., Simonyan, K., Elsen, E., Rae, J.W., Vinyals, O., Sifre, L.: Training compute-optimal large language models. In: Advances in Neural Information Processing Systems. vol. 35 (2022). https://doi.org/10.48550/arXiv.2203.15556, `https://proceedings.neurips.cc/paper/2022/hash/c1e2faff6f588870935f114ebe04a3e5-Abstract-Conference.html`

[60] Hosseini-Asl, E., McCann, B., Wu, C.S., Yavuz, S., Socher, R.: A simple language model for task-oriented dialogue. In: Advances in Neural Information Processing Systems (NeurIPS). vol. 33 (2020). https://doi.org/10.48550/arXiv.2005.00796, `https://proceedings.neurips.cc/paper_files/paper/2020/hash/e946209592563be0f01c844ab2170f0c-Abstract.html`

[61] Husain, H., Wu, H., Gazit, T., Allamanis, M., Brockschmidt, M.: Codesearchnet challenge: Evaluating the state of semantic code search. ArXiv **abs/1909.09436** (2019). https://doi.org/10.48550/arXiv.1909.09436, `https://arxiv.org/abs/1909.09436`

[62] Jelinek, F.: Self-organized language modeling for speech recognition, pp. 450–506. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1990). https://doi.org/10.5555/108235.108270, `https://dl.acm.org/doi/10.5555/108235.108270`

[63] Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y., Chen, D., Dai, W., Madotto, A., Fung, P.: Survey of hallucination in natural language generation. ACM Computing Surveys **55**(12), 1–38 (2023). https://doi.org/10.1145/3571730, `https://dl.acm.org/doi/10.1145/3571730`

[64] Jiang, Z., Mao, Y., He, P., Neubig, G., Chen, W.: Omnitab: Pretraining with natural and synthetic data for few-shot table-based question answering. In: Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. pp. 932–942 (2022). https://doi.org/10.18653/v1/2022.naacl-main.68, `https://aclanthology.org/2022.naacl-main.68/`

[65] Jones, K.S., Walker, S., Robertson, S.E.: A probabilistic model of information retrieval: development and comparative experiments: Part 2. Information processing & management **36**(6), 809–840 (2000)

[66] Kaplan, J., McCandlish, S., Henighan, T.J., Brown, T.B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., Amodei, D.: Scaling laws for neural language models. arXiv preprint **abs/2001.08361** (2020). https://doi.org/10.48550/arXiv.2001.08361, `https://arxiv.org/abs/2001.08361`, no peer-reviewed venue; arXiv only

[67] Karpukhin, V., Oğuz, B., Min, S., Lewis, P., Wu, L.Y., Edunov, S., Chen, D., tau Yih, W.: Dense passage retrieval for open-domain question answering. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP). pp. 6769–6781. Association for Computational Linguistics, Online (2020). https://doi.org/10.18653/v1/2020.emnlp-main.550, `https://aclanthology.org/2020.emnlp-main.550/`

[68] Kate, K., Pedapati, T., Basu, K., Rizk, Y., Chenthamarakshan, V., Chaudhury, S., Agarwal, M., Abdelaziz, I.: LongFuncEval: Measuring the effectiveness of long context models for function calling. arXiv preprint arXiv:2505.10570 (2025)

[69] Keysers, D., Schärli, N., Scales, N., Buisman, H., Furrer, D., Kashubin, S., Momchev, N., Sinopalnikov, D., Stafiniak, L., Tihon, T., Tsarkov, D., Wang, X., van Zee, M., Bousquet, O.: Measuring compositional generalization: A comprehensive method on realistic data. In: International Conference on Learning Representations (2020). https://doi.org/10.48550/arXiv.1912.09713, `https://openreview.net/forum?id=SygcCnNKwr`

[70] Khashabi, D., Min, S., Khot, T., Sabharwal, A., Tafjord, O., Clark, P., Hajishirzi, H.: UNIFIEDQA: Crossing format boundaries with a single QA system. In: Findings of the Association for Computational Linguistics: EMNLP 2020. pp. 1896–1907. Association for Computational Linguistics, Online (2020). https://doi.org/10.18653/v1/2020.findings-emnlp.171, `https://aclanthology.org/2020.findings-emnlp.171/`

[71] Kim, S., Yang, S., Kim, G., Lee, S.W.: Efficient dialogue state tracking by selectively overwriting memory. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. pp. 567–582 (2020). https://doi.org/10.18653/v1/2020.acl-main.53, `https://aclanthology.org/2020.acl-main.53/`

[72] Kneser, R., Ney, H.: Improved backing-off for m-gram language modeling. In: 1995 International Conference on Acoustics, Speech, and Signal Processing (ICASSP). vol. 1, pp. 181–184 (1995). https://doi.org/10.1109/ICASSP.1995.479394, `https://ieeexplore.ieee.org/document/479394`

[73] Kojima, T., Gu, S.S., Reid, M., Matsuo, Y., Iwasawa, Y.: Large language models are zero-shot reasoners. In: Advances in Neural Information Processing Systems. vol. 35 (2022). https://doi.org/10.5555/3600270.3601883,

https://proceedings.neurips.cc/paper_files/paper/2022/hash/8bb0d291acd4acf06ef112099c16f326-Abstract-Conference.html

[74] Komeili, M., Shuster, K., Weston, J.: Internet-augmented dialogue generation. In: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 8460–8478 (2022). https://doi.org/10.18653/v1/2022.acl-long.579, https://aclanthology.org/2022.acl-long.579/

[75] Lake, B.M., Baroni, M.: Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In: Proceedings of the 35th International Conference on Machine Learning. pp. 2873–2882 (2018). https://doi.org/10.48550/arXiv.1711.00350, https://proceedings.mlr.press/v80/lake18a.html

[76] Lavrenko, V., Croft, W.B.: Relevance-based language models. In: Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 120–127. ACM, New Orleans, Louisiana, USA (2001). https://doi.org/10.1145/383952.383972, https://doi.org/10.1145/383952.383972

[77] van Lawick-Goodall, J.: Tool-using in primates and other vertebrates. Advances in The Study of Behavior **3**, 195–249 (1971). https://doi.org/10.1016/S0065-3454(08)60157-6, https://doi.org/10.1016/S0065-3454(08)60157-6

[78] Lazaridou, A., Gribovskaya, E., Stokowiec, W., Grigorev, N.: Internet-augmented language models through few-shot prompting for open-domain question answering. ArXiv **abs/2203.05115** (2022). https://doi.org/10.48550/arXiv.2203.05115, https://arxiv.org/abs/2203.05115

[79] LeCun, Y., Bengio, Y., Hinton, G.E.: Deep learning. Nature **521**(7553), 436–444 (2015). https://doi.org/10.1038/nature14539, https://doi.org/10.1038/nature14539

[80] Lei, W., Jin, X., Kan, M.Y., Ren, Z., He, X., Yin, D.: Sequicity: Simplifying task-oriented dialogue systems with single sequence-to-sequence architectures. In: Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 1437–1447 (2018). https://doi.org/10.18653/v1/P18-1133, https://aclanthology.org/P18-1133/

[81] Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., rahman Mohamed, A., Levy, O., Stoyanov, V., Zettlemoyer, L.: Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In: Proceedings of the

58th Annual Meeting of the Association for Computational Linguistics. pp. 7871–7880 (2020). https://doi.org/10.18653/v1/2020.acl-main.703, `https://aclanthology.org/2020.acl-main.703/`

[82] Li, R., Allal, L.B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., Liu, Q., Zheltonozhskii, E., Zhuo, T.Y., Wang, T., Dehaene, O., Davaadorj, M., Lamy-Poirier, J., Monteiro, J., Shliazhko, O., et al.: StarCoder: may the source be with you! arXiv preprint arXiv:2305.06161 (2023)

[83] Liang, J., Huang, W., Xia, F., Xu, P., Hausman, K., Ichter, B., Florence, P.R., Zeng, A.: Code as policies: Language model programs for embodied control. In: 2023 IEEE International Conference on Robotics and Automation (ICRA). pp. 9493–9500 (2023). https://doi.org/10.1109/ICRA55551.2023.10160591, `https://ieeexplore.ieee.org/document/10160591`

[84] Liang, P.: Semi-Supervised Learning for Natural Language. Master's thesis, Massachusetts Institute of Technology (2005), `https://people.csail.mit.edu/pliang/papers/meng-thesis.pdf`

[85] Lin, S., Hilton, J., Evans, O.: Teaching models to express their uncertainty in words. Transactions on Machine Learning Research (2022). https://doi.org/10.48550/arXiv.2205.14334, `https://openreview.net/forum?id=8s8K2UZGTZ`

[86] Ling, W., Blunsom, P., Grefenstette, E., Hermann, K.M., Kociský, T., Wang, F., Senior, A.W.: Latent predictor networks for code generation. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 599–609. Association for Computational Linguistics, Berlin, Germany (2016). https://doi.org/10.18653/v1/P16-1057, `https://aclanthology.org/P16-1057/`

[87] Liu, N.F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., Liang, P.: Lost in the middle: How language models use long contexts. Transactions of the Association for Computational Linguistics **12**, 157–173 (2024). https://doi.org/10.1162/tacl_a_00638, `https://aclanthology.org/2024.tacl-1.9/`

[88] Liu, Q., Chen, B., Guo, J., Lin, Z., Lou, J.G.: TAPEX: Table pre-training via learning a neural SQL executor. In: International Conference on Learning Representations (2022). https://doi.org/10.48550/arXiv.2107.07653, `https://openreview.net/forum?id=O50443AsCP`

[89] Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., Stoyanov, V.: Roberta: A robustly optimized bert pretraining approach. In: International Conference on Learning Representations (ICLR) (2020). https://doi.org/10.48550/arXiv.1907.11692, `https://openreview.net/forum?id=SyxS0T4tvS`

[90] Maravita, A., Iriki, A.: Tools for the body (schema). Trends in Cognitive Sciences **8**(2), 79–86 (2004). https://doi.org/10.1016/j.tics.2003.12.008, `https://doi.org/10.1016/j.tics.2003.12.008`

[91] Maynez, J., Narayan, S., Bohnet, B., McDonald, R.T.: On faithfulness and factuality in abstractive summarization. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. pp. 1906–1919. Association for Computational Linguistics, Online (2020). https://doi.org/10.18653/v1/2020.acl-main.173, `https://aclanthology.org/2020.acl-main.173/`

[92] McCulloch, W.S., Pitts, W.: A logical calculus of the ideas immanent in nervous activity. Bulletin of Mathematical Biophysics **5**, 115–133 (1943). https://doi.org/10.1007/BF02478259, `https://doi.org/10.1007/BF02478259`

[93] Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J.: Distributed representations of words and phrases and their compositionality. In: Advances in Neural Information Processing Systems. vol. 26 (2013). https://doi.org/10.48550/arXiv.1310.4546, `https://proceedings.neurips.cc/paper/2013/hash/9aa42b31882ec039965f3c4923ce901b-Abstract.html`

[94] Mishne, G., de Rijke, M.: Source code retrieval using conceptual similarity. In: RIAO 2004: Coupling Approaches, Coupling Media and Coupling Languages for Information Retrieval (2004), `https://staff.fnwi.uva.nl/m.derijke/wp-content/papercite-data/pdf/mishne-source-2004.pdf`

[95] Mrksic, N., Séaghdha, D.Ó., Wen, T.H., Thomson, B., Young, S.J.: Neural belief tracker: Data-driven dialogue state tracking. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 1777–1788 (2017). https://doi.org/10.18653/v1/P17-1163, `https://aclanthology.org/P17-1163/`

[96] Neelakantan, A., Le, Q.V., Sutskever, I.: Neural programmer: Inducing latent programs with gradient descent. In: International Conference on Learning Representations (2016). https://doi.org/10.48550/arXiv.1511.04834, `https://arxiv.org/abs/1511.04834`

[97] Nogueira, R., Cho, K.: Passage re-ranking with bert. ArXiv **abs/1901.04085** (2019). https://doi.org/10.48550/arXiv.1901.04085, `https://arxiv.org/abs/1901.04085`

[98] Nogueira, R., Jiang, Z., Lin, J.: Investigating the limitations of transformers with simple arithmetic tasks (2021). https://doi.org/10.48550/arXiv.2102.13019, `https://arxiv.org/abs/2102.13019`

[99] Nogueira, R., Yang, W., Cho, K., Lin, J.J.: Multi-stage document ranking with bert. ArXiv **abs/1910.14424** (2019). https://doi.org/10.48550/arXiv.1910.14424, `https://arxiv.org/abs/1910.14424`

[100] Nye, M., Andreassen, A., Gur-Ari, G., Michalewski, H., Austin, J., Bieber, D., Dohan, D., Lewkowycz, A., Bosma, M., Luan, D., Sutton, C., Odena, A.: Show your work: Scratchpads for intermediate computation with language models. In: International Conference on Learning Representations (ICLR) (2022). https://doi.org/10.48550/arXiv.2112.00114, `https://openreview.net/forum?id=iedYJm92o0a`

[101] Oda, Y., Fudaba, H., Neubig, G., Hata, H., Sakti, S., Toda, T., Nakamura, S.: Learning to generate pseudo-code from source code using statistical machine translation (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 574–584 (2015). https://doi.org/10.1109/ASE.2015.36, `https://ieeexplore.ieee.org/document/7372045`

[102] Orban, G.A., Caruana, F.: The neural basis of human tool use. Frontiers in Psychology **5**, 310 (2014). https://doi.org/10.3389/fpsyg.2014.00310, `https://www.frontiersin.org/articles/10.3389/fpsyg.2014.00310/full`

[103] Papineni, K., Roukos, S., Ward, T., Zhu, W.J.: Bleu: a method for automatic evaluation of machine translation. In: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics. pp. 311–318 (2002). https://doi.org/10.3115/1073083.1073135, `https://aclanthology.org/P02-1040/`

[104] Parisi, A., Zhao, Y., Fiedel, N.: TALM: Tool augmented language models (2022). https://doi.org/10.48550/arXiv.2205.12255, `https://arxiv.org/abs/2205.12255`

[105] Pascanu, R., Mikolov, T., Bengio, Y.: On the difficulty of training recurrent neural networks. In: Proceedings of the 30th International Conference on Machine Learning. vol. 28, pp. 1310–1318. PMLR (2013). https://doi.org/10.5555/3042817.3043083, `https://dl.acm.org/doi/10.5555/3042817.3043083`

[106] Pasupat, P., Liang, P.: Compositional semantic parsing on semi-structured tables. In: Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers). pp. 1470–1480. Association for Computational Linguistics, Beijing, China (2015). https://doi.org/10.3115/v1/P15-1142, `https://aclanthology.org/P15-1142/`

[107] Patel, A., Bhattamishra, S., Goyal, N.: Are NLP models really able to solve simple math word problems? In: Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. pp. 2080–2094. Association for Computational Linguistics, Online (2021). https://doi.org/10.18653/v1/2021.naacl-main.168, `https://aclanthology.org/2021.naacl-main.168/`

[108] Pennington, J., Socher, R., Manning, C.D.: Glove: Global vectors for word representation. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP). pp. 1532–1543 (2014). https://doi.org/10.3115/v1/D14-1162, `https://aclanthology.org/D14-1162/`

[109] Peters, M.E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., Zettlemoyer, L.: Deep contextualized word representations. In: Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers). pp. 2227–2237 (2018). https://doi.org/10.18653/v1/N18-1202, `https://aclanthology.org/N18-1202/`

[110] Petroni, F., Rocktäschel, T., Lewis, P., Bakhtin, A., Wu, Y., Miller, A.H., Riedel, S.: Language models as knowledge bases? In: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP). pp. 2463–2473. Association for Computational Linguistics, Hong Kong, China (2019). https://doi.org/10.18653/v1/D19-1250, `https://aclanthology.org/D19-1250/`

[111] Rabinovich, M., Stern, M., Klein, D.: Abstract syntax networks for code generation and semantic parsing. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 1139–1149 (2017). https://doi.org/10.18653/v1/P17-1105, `https://aclanthology.org/P17-1105/`

[112] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I.: Language models are unsupervised multitask learners (2019), `https://cdn.openai.com/`

better-language-models/language_models_are_unsupervised_multitask_learners.pdf

[113] Rae, J.W., Borgeaud, S., Cai, T., Millican, K., Hoffmann, J., Song, F., Aslanides, J., Henderson, S., Ring, R., Young, S., Rutherford, E., Hennigan, T., Menick, J., Cassirer, A., Powell, R., van den Driessche, G., Hendricks, L.A., Rauh, M., Huang, P.S., Glaese, A., Welbl, J., Dathathri, S., Huang, S., Uesato, J., Mellor, J.F.J., Higgins, I., Creswell, A., McAleese, N., Wu, A., Elsen, E., Jayakumar, S.M., Buchatskaya, E., Budden, D., Sutherland, E., Simonyan, K., Paganini, M., Sifre, L., Martens, L., Li, X.L., Kuncoro, A., Nematzadeh, A., Gribovskaya, E., Donato, D., Lazaridou, A., Mensch, A., Lespiau, J.B., Tsimpoukelli, M., Grigorev, N.K., Fritz, D., Sottiaux, T., Pajarskas, M., Pohlen, T., Gong, Z., Toyama, D., de Masson d'Autume, C., Li, Y., Terzi, T., Mikulik, V., Babuschkin, I., Clark, A., de Las Casas, D., Guy, A., Jones, C., Bradbury, J., Johnson, M.G., Hechtman, B.A., Weidinger, L., Gabriel, I., Isaac, W.S., Lockhart, E., Osindero, S., Rimell, L., Dyer, C., Vinyals, O., Ayoub, K.W., Stanway, J., Bennett, L.L., Hassabis, D., Kavukcuoglu, K., Irving, G.: Scaling language models: Methods, analysis & insights from training gopher. ArXiv **abs/2112.11446** (2021). https://doi.org/10.48550/arXiv.2112.11446, https://arxiv.org/abs/2112.11446

[114] Raffel, C., Shazeer, N.M., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., Liu, P.J.: Exploring the limits of transfer learning with a unified text-to-text transformer. Journal of Machine Learning Research **21**(140), 1–67 (2020). https://doi.org/10.5555/3455716.3455856, https://jmlr.org/papers/v21/20-074.html

[115] Razumovskaia, E., Glavavs, G., Majewska, O., Ponti, E., Korhonen, A., Vulic, I.: Crossing the conversational chasm: A primer on natural language processing for multilingual task-oriented dialogue systems. Journal of Artificial Intelligence Research **74**, 1351–1402 (2022). https://doi.org/10.1613/jair.1.13083, https://jair.org/index.php/jair/article/view/13083

[116] Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X.E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C.C., Grattafiori, A., Xiong, W., Défossez, A., Chaudhary, V., et al.: Code Llama: Open foundation models for code. arXiv preprint arXiv:2308.12950 (2023)

[117] Ruder, S.: An overview of gradient descent optimization algorithms. ArXiv **abs/1609.04747** (2016). https://doi.org/10.48550/arXiv.1609.04747, https://arxiv.org/abs/1609.04747

[118] Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back-propagating errors. Nature **323**, 533–536 (1986). https://doi.org/10.1038/323533a0, `https://www.nature.com/articles/323533a0`

[119] Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., Scialom, T.: Toolformer: Language models can teach themselves to use tools. In: Advances in Neural Information Processing Systems (NeurIPS). vol. 36 (2023). https://doi.org/10.48550/arXiv.2302.04761, `https://proceedings.neurips.cc/paper_files/paper/2023/hash/d842425e4bf79ba039352da0f658a906-Abstract-Conference.html`

[120] Schmidhuber, J.: Towards compositional learning in dynamic networks. Tech. Rep. FKI-129-90, Technische Universität München (1990), `https://people.idsia.ch/~juergen/FKI-129-90ocr.pdf`

[121] See, A., Liu, P.J., Manning, C.D.: Get to the point: Summarization with pointer-generator networks. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 1073–1083. Association for Computational Linguistics, Vancouver, Canada (2017). https://doi.org/10.18653/v1/P17-1099, `https://aclanthology.org/P17-1099/`

[122] Sellam, T., Das, D., Parikh, A.P.: Bleurt: Learning robust metrics for text generation. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. pp. 7881–7892. Association for Computational Linguistics (2020). https://doi.org/10.18653/v1/2020.acl-main.704, `https://aclanthology.org/2020.acl-main.704/`

[123] Sennrich, R., Haddow, B., Birch, A.: Neural machine translation of rare words with subword units. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 1715–1725. Association for Computational Linguistics, Berlin, Germany (2016). https://doi.org/10.18653/v1/P16-1162, `https://aclanthology.org/P16-1162/`

[124] Shannon, C.E.: A mathematical theory of communication. Bell Syst. Tech. J. **27**, 623–656 (1948). https://doi.org/10.1002/j.1538-7305.1948.tb01338.x, `https://doi.org/10.1002/j.1538-7305.1948.tb01338.x`

[125] Shi, T., Zhao, C., Boyd-Graber, J.L., Daum'e, H., Lee, L.: On the potential of lexico-logical alignments for semantic parsing to sql queries. In: Findings of the Association for Computational Linguistics: EMNLP 2020. pp. 1849–1864 (2020). https://doi.org/10.18653/v1/2020.findings-emnlp.167, `https://aclanthology.org/2020.findings-emnlp.167/`

[126] Shin, R., Lin, C.H., Thomson, S., Chen, C.C., Roy, S., Platanios, E.A., Pauls, A., Klein, D., Eisner, J., Durme, B.V.: Constrained language models yield few-shot semantic parsers. In: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing. pp. 7699–7715 (2021). https://doi.org/10.18653/v1/2021.emnlp-main.608, `https://aclanthology.org/2021.emnlp-main.608/`

[127] Sparck Jones, K.: A statistical interpretation of term specificity and its application in retrieval. Journal of documentation **28**(1), 11–21 (1972)

[128] hao Su, P., Gasic, M., Mrksic, N., Rojas-Barahona, L.M., Ultes, S., Vandyke, D., Wen, T.H., Young, S.J.: On-line active reward learning for policy optimisation in spoken dialogue systems. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 2431–2441 (2016). https://doi.org/10.18653/v1/P16-1230, `https://aclanthology.org/P16-1230/`

[129] Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: Advances in Neural Information Processing Systems 27 (NeurIPS 2014). pp. 3104–3112. Curran Associates, Inc. (2014). https://doi.org/10.48550/arXiv.1409.3215, `https://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks`

[130] Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., Lample, G.: Llama: Open and efficient foundation language models. ArXiv **abs/2302.13971** (2023). https://doi.org/10.48550/arXiv.2302.13971, `https://arxiv.org/abs/2302.13971`

[131] Touvron, H., Martin, L., Stone, K.R., Albert, P., Almahairi, A., Babaei, Y., lay Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D.M., Blecher, L., tian Cantón Ferrer, C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A.S., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I.M., Korenev, A., Koura, P.S., Lachaux, M.A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E.M., Subramanian, R., Tan, X., Tang, B., Taylor, R., Williams, A., Kuan, J.X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M.H.M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., Scialom, T.: Llama 2: Open foundation and fine-tuned chat models. ArXiv **abs/2307.09288** (2023). https://doi.org/10.48550/arXiv.2307.09288, `https://arxiv.org/abs/2307.09288`

[132] Ueffing, N., Ney, H.: Word-level confidence estimation for machine translation using phrase-based translation models. In: Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing. pp. 763–770 (2005), `https://aclanthology.org/H05-1096/`

[133] Vaswani, A., Shazeer, N.M., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. In: Advances in Neural Information Processing Systems. vol. 30 (2017). https://doi.org/10.48550/arXiv.1706.03762, `https://proceedings.neurips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html`

[134] Vinyals, O., Fortunato, M., Jaitly, N.: Pointer networks. In: Advances in Neural Information Processing Systems (NeurIPS). vol. 28 (2015). https://doi.org/10.48550/arXiv.1506.03134, `https://proceedings.neurips.cc/paper_files/paper/2015/hash/29921001f2f04bd3baee84a12e98098f-Abstract.html`

[135] Wei, J., Bosma, M., Zhao, V.Y., Guu, K., Yu, A.W., Lester, B., Du, N., Dai, A.M., Le, Q.V.: Finetuned language models are zero-shot learners. In: International Conference on Learning Representations (2022). https://doi.org/10.48550/arXiv.2109.01652, `https://openreview.net/forum?id=gEZrGCozdqR`

[136] Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., Chi, E.H., Hashimoto, T., Vinyals, O., Liang, P., Dean, J., Fedus, W.: Emergent abilities of large language models. Transactions on Machine Learning Research (2022). https://doi.org/10.48550/arXiv.2206.07682, `https://openreview.net/forum?id=yzkSU5zdwD`

[137] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E.H., Le, Q.V., Zhou, D.: Chain-of-thought prompting elicits reasoning in large language models. In: Advances in Neural Information Processing Systems. vol. 35 (2022). https://doi.org/10.48550/arXiv.2201.11903, `https://proceedings.neurips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html`

[138] Welleck, S., West, P., Cao, J., Choi, Y.: Symbolic brittleness in sequence models: on systematic generalization in symbolic mathematics. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 36, pp. 8495–8503 (2022). https://doi.org/10.1609/aaai.v36i8.20841, `https://ojs.aaai.org/index.php/AAAI/article/view/20841`

[139] Williams, J., Asadi, K., Zweig, G.: Hybrid code networks: practical and efficient end-to-end dialog control with supervised and reinforcement learning. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 665–677 (2017). https://doi.org/10.18653/v1/P17-1062, `https://aclanthology.org/P17-1062/`

[140] Williams, J., Raux, A., Henderson, M.: The dialog state tracking challenge series: A review. Dialogue & Discourse **7**, 4–33 (2016). https://doi.org/10.5087/dad.2016.301, `https://doi.org/10.5087/dad.2016.301`

[141] Williams, J., Young, S.J.: Partially observable markov decision processes for spoken dialog systems. Computer Speech and Language **21**, 393–422 (2007). https://doi.org/10.1016/j.csl.2006.06.008, `https://doi.org/10.1016/j.csl.2006.06.008`

[142] Wolfson, T., Geva, M., Gupta, A., Gardner, M., Goldberg, Y., Deutch, D., Berant, J.: Break it down: A question understanding benchmark. Transactions of the Association for Computational Linguistics **8**, 183–198 (2020). https://doi.org/10.1162/tacl_a_00309, `https://aclanthology.org/2020.tacl-1.13/`

[143] Woods, W.A.: Progress in natural language understanding: An application to lunar geology. In: Proceedings of the National Computer Conference and Exposition. pp. 441–450 (1973). https://doi.org/10.1145/1499586.1499695, `https://dl.acm.org/doi/10.1145/1499586.1499695`

[144] Wu, Y., Schuster, M., Chen, Z., Le, Q.V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, L., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J.R., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G.S., Hughes, M., Dean, J.: Google's neural machine translation system: Bridging the gap between human and machine translation (2016). https://doi.org/10.48550/arXiv.1609.08144, `https://arxiv.org/abs/1609.08144`

[145] Xie, T., Wu, C.H., Shi, P., Zhong, R., Scholak, T., Yasunaga, M., Wu, C.S., Zhong, M., Yin, P., Wang, S.I., Zhong, V., Wang, B., Li, C., Boyle, C., Ni, A., Yao, Z., Radev, D.R., Xiong, C., Kong, L., Zhang, R., Smith, N.A., Zettlemoyer, L., Yu, T.: Unifiedskg: Unifying and multi-tasking structured knowledge grounding with text-to-text language models. In: Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing. pp. 602–631 (2022). https://doi.org/10.18653/v1/2022.emnlp-main.39, `https://aclanthology.org/2022.emnlp-main.39/`

[146] Xu, F.F., Jiang, Z., Yin, P., Vasilescu, B., Neubig, G.: Incorporating external knowledge through pre-training for natural language to code generation. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. pp. 6045–6052 (2020). https://doi.org/10.18653/v1/2020.acl-main.538, `https://aclanthology.org/2020.acl-main.538/`

[147] Yang, Y., Li, Y., Quan, X.: Ubar: Towards fully end-to-end task-oriented dialog systems with gpt-2. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 35, pp. 13268–13276 (2021). https://doi.org/10.1609/aaai.v35i15.17674, `https://ojs.aaai.org/index.php/AAAI/article/view/17674`

[148] Yin, P., Deng, B., Chen, E., Vasilescu, B., Neubig, G.: Learning to mine aligned code and natural language pairs from stack overflow. In: Proceedings of the 15th International Conference on Mining Software Repositories (MSR). pp. 476–486. ACM (2018). https://doi.org/10.1145/3196398.3196408, `https://dl.acm.org/doi/10.1145/3196398.3196408`, iEEE/ACM International Conference on Mining Software Repositories

[149] Yin, P., Fang, H., Neubig, G., Pauls, A., Platanios, E.A., Su, Y., Thomson, S., Andreas, J.: Compositional generalization for neural semantic parsing via span-level supervised attention. In: Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. pp. 2810–2823. Association for Computational Linguistics, Online (2021). https://doi.org/10.18653/v1/2021.naacl-main.225, `https://aclanthology.org/2021.naacl-main.225/`

[150] Yin, P., Neubig, G.: TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. In: Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations. pp. 7–12. Association for Computational Linguistics, Brussels, Belgium (2018). https://doi.org/10.18653/v1/D18-2002, `https://aclanthology.org/D18-2002/`

[151] Yin, P., Neubig, G., tau Yih, W., Riedel, S.: Tabert: Pretraining for joint understanding of textual and tabular data. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. pp. 8413–8426 (2020). https://doi.org/10.18653/v1/2020.acl-main.745, `https://aclanthology.org/2020.acl-main.745/`

[152] Young, S.J., Gasic, M., Thomson, B., Williams, J.D.: POMDP-based statistical spoken dialog systems: A review. Proceedings of the IEEE **101**(5),

1160–1179 (2013). https://doi.org/10.1109/JPROC.2012.2225812, `https://ieeexplore.ieee.org/document/6407655/`

[153] Zaremba, W., Sutskever, I.: Learning to execute. ArXiv **abs/1410.4615** (2014). https://doi.org/10.48550/arXiv.1410.4615, `https://arxiv.org/abs/1410.4615`

[154] Zelle, J.M., Mooney, R.J.: Learning to parse database queries using inductive logic programming. In: Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96). pp. 1050–1055 (1996), `https://cdn.aaai.org/AAAI/1996/AAAI96-156.pdf`

[155] Zettlemoyer, L., Collins, M.: Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In: Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence (UAI 2005). pp. 658–666 (2005). https://doi.org/10.48550/arXiv.1207.1420, `https://arxiv.org/abs/1207.1420`

[156] Zhong, V., Xiong, C., Socher, R.: Seq2sql: Generating structured queries from natural language using reinforcement learning. In: International Conference on Learning Representations (2018). https://doi.org/10.48550/arXiv.1709.00103, `https://openreview.net/forum?id=Syx6bz-Ab`

[157] Zhou, D., Scharli, N., Hou, L., Wei, J., Scales, N., Wang, X., Schuurmans, D., Bousquet, O., Le, Q., Chi, E.H.: Least-to-most prompting enables complex reasoning in large language models. In: The Eleventh International Conference on Learning Representations (2023). https://doi.org/10.48550/arXiv.2205.10625, `https://openreview.net/forum?id=WZH7099tgfM`