Generating Code for Chemistry Simulations using Large Language Models

Pieter Floris Jacobs (s3777693)

June 17, 2024

Internal Supervisor: Dr. Matthia Sabatelli (Artificial Intelligence, University of Groningen)
External Supervisor: Dr. Robert Pollice (Chemistry, University of Groningen)

Artificial Intelligence
University of Groningen, The Netherlands

Abstract

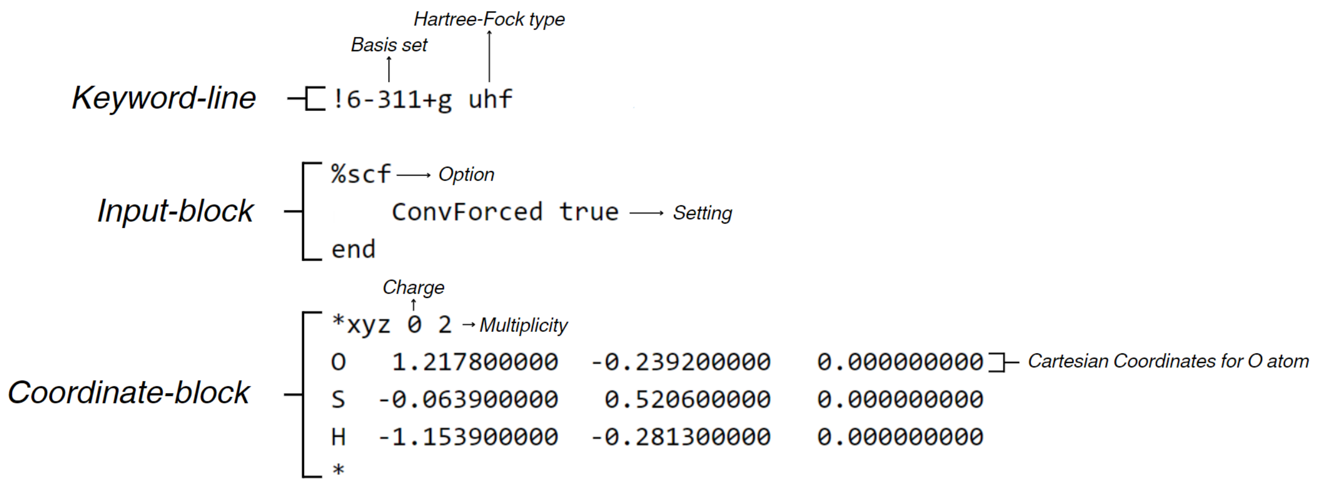
1 Introduction

Large language models (LLMs), like GPT-4 (75) and Gemini 1.5 (100), are advanced NLP systems that are characterised by their massive sets of trainable parameters, often in the range of billions. They are typically pre-trained on vast, general-purpose corpora that encompass diverse text formats, including news articles, code snippets, and even social media interactions to give them a general understanding of natural language. Through giving them task-specific instructions, referred to as prompts, they have been shown to use their pre-trained knowledge to solve various complex tasks. One of these tasks is code synthesis, where LLMs translate natural language instructions into functional code for various programming languages like Python or Java (17). The advent of code synthesis has the potential to boost the efficiency of software development. Moreover, it can make programming tools more accessible to individuals with minimal programming experience, thereby boosting overall productivity.

While LLMs show impressive capabilities in general purpose programming languages, they can struggle with domain-specific languages (DSLs) (34). DSLs are specialized by definition and are thus unlikely to have been encountered often enough during pretraining for the LLM to acquire a deep enough understanding of the language, especially its syntax and semantics. One could argue that this limitation becomes particularly critical for DSLs because researchers in specific domains could significantly enhance their productivity by leveraging LLMs for programming tasks. By doing so, they can streamline their programming efforts and allocate more time and energy to their core research activities, where their expertise lies.

Chemistry is an example of a research field in which LLMs have already been used in various ways, like for molecule editing (62) or molecular property prediction (112; 19; 2). Their ability to write Python code to solve chemistry problems has been assessed by White et al. (119), with some promising results. However, an area yet to be extensively explored is creating an LLM specifically for generating code for chemistry-specific physics-based simulation packages. Creating an LLM adept at generating chemistry simulations could assist researchers in their efforts of creating working simulations by alleviating the need for large familiarity with specific simulation packages. Even for experts that are well-versed in the DSL of a given package, significant challenges can arise in efficiently navigating and crafting simulation code. The LLM's chat-like interface would offer them a way to seek clarification, troubleshoot issues, and deepen their understanding of the simulation its principles.

These computational chemistry software packages, such as Gaussian (37), PySCF (98), and ORCA (69), are rooted in theoretical physics principles. They use mathematical models and algorithms derived from branches like quantum- and classical mechanics to predict properties, behaviors, and interactions of molecules and materials by describing the behavior of electrons and atoms within molecules. For example, one can use them for automatic calculation of the Hamiltonian, which encapsulates the total energy of a system in terms of the positions and momenta of its constituent particles, enabling precise simulations of molecular behavior. These packages facilitate a wide range of such complex calculations, if they are provided with a syntactically and semantically correct input file. There is no research available about direct code synthesis for the mentioned packages. However, since PySCF is a python library,



```
!6-311+g uhf

%scf
    ConvForced true
end

*xyz 0 2
O    1.217800000  -0.239200000   0.000000000
S   -0.063900000   0.520600000   0.000000000
H   -1.153900000  -0.281300000   0.000000000
*
```

Figure 1: An ORCA input file consisting for an unrestricted hartree-fock calculation.

one could argue that the LLMs should be able to perform well or have less to learn, as code-synthesis LLMs perform well on python tasks. As for Gaussian, Hocky and White (44) has explored indirect code synthesis for Gaussian. They used the Codex (17) model to create python functions to generate Gaussian input files. As of yet, no research has looked into using LLMs for code synthesis for ORCA.

ORCA has been around since 1990, and has seen active development until now (69). It is known for its ease of use. This is largely because it has comprehensive documentation and robust error handling to help users identify and resolve issues that may arise during calculation together with an output file containing a thorough description of the calculations that are performed. These pieces of information are useful for an LLM: documentation can be used for teaching the LLM about the language and the warnings can be used in a feedback loop. ORCA is powerful, and can run a wide range of different calculations while at the same time being free to use for personal or academic purposes. Therefore, together with its solid standing in the computational and theoretical chemistry community, this study has opted to create an LLM for ORCA over other available packages. Like mentioned earlier, to work with ORCA, its user needs to provide it with a syntactically and semantically correct input file. An example of a simple ORCA input file is provided in Figure 1. It typically contains keyword lines, starting with "!" and defining global calculation options; input blocks, enclosed between % and end, offering fine control over specific settings; and a coordinate block, enclosed within * symbols, specifying the molecular geometry, charge, and multiplicity. Comments start with # and are ignored by the program.

Synthesising such an input file from natural language could, as previously mentioned, boost efficiency of both inexperienced and experienced users alike. This study will therefore focus on code synthesis for ORCA input files. More specifically: given a pre-trained LLM, how can we optimally modify it to understand a DSL like ORCA's input? Conceptually, answering this question requires us to find a way to optimally exploit the already available knowledge of the LLM and teaching the LLM new unknown knowledge that is essential for understanding the DSL in question.

An essential part of leveraging the LLM's pre-trained understanding of language lies in crafting effective ways to prompt it. These prompts, which serve as specific instructions or questions that are the input provided to the model, guide the LLM towards generating the desired output. Prompt engineering has received a lot of attention in recent literature (15) because small changes to the prompt, such as its phrasing and the sequence of examples incorporated within it, can result in very significant changes in the output (48; 65). Research has found various effective methodologies that deal with how to format an effective prompt, including Chain-of-Thought (CoT) (116) prompting, Tree-of-Thoughts (ToT) and Graph-of-Thoughts (GoT) prompting.

Even though reformatting a prompt can help a model better understand and generate a more accurate response, it does not provide new information to the model. LLMs have a limited context-window, so one can generally not include a full database of relevant information to the model. Therefore, research has focused on how one can give an LLM optimal access to external information (77; 52; 111). A prevalent example for this principle is Retrieval Augmented Generation (RAG) (39). RAG functions by allowing the LLM to access and leverage relevant information from external knowledge sources beyond its pre-training data. This can, for instance, be achieved by embedding relevant textual context sources, and using similarity search between the user-prompt and this database to retrieve relevant pieces of text. To exemplify this even further, in the context of ORCA code synthesis, this could involve retrieving examples of correct ORCA syntax and usage from online repositories or documentation. By incorporating this retrieved information alongside the user prompt and the LLM's pre-trained understanding of language, RAG could potentially enable the LLM to generate more accurate and syntactically correct ORCA input files. This can be particularly beneficial for complex calculations (57) or simply when the user's natural language description is ambiguous (53).

Despite the considerable benefits of optimizing prompts, this approach may fall short in achieving the desired level of accuracy when dealing with a DSL. As the LLM has limited training exposure to DSL-specific data during pre-training, there is supposedly little knowledge about the general structure of a language available to exploit with prompting. A variety of studies have shown that so-called fine-tuning can improve task-specific performance of language models (122; 109). Fine-tuning is a form of transfer learning (118), where the knowledge gained from pre-training on a large generic dataset is transferred and refined for a specific task or domain. The model is trained on a curated dataset containing examples of desired inputs and outputs, such as input files and their corresponding natural language descriptions like in our case of teaching the model how to generate ORCA input files. This process can allow the model to adapt its parameters to better suit the nuances and intricacies of the target domain (i.e., computational chemistry) and task (i.e., ORCA code synthesis). Note, however, that Gekhman et al. (41) found that when factual knowledge is new to an LLM, like in our case ORCA-specific terms used in an input file, it is learned significantly slower using finetuning when compared to pre-training and may even increase the amount of hallucinations (123) the finetuned LLM produces.

Aside from this, finetuning presents another significant challenge: the acquisition of a suitable dataset. This is especially problematic for our case of ORCA code synthesis, due to the specialized nature of the required data. Generally, there is little data available for DSLs, hence our initial problem of LLMs currently not being able to generalise towards them. Using a dataset of limited size might result in the model only experiencing minor adjustments to its pre-trained parameters, potentially limiting its ability to effectively adapt to the specificity of the DSL. It have previously been shown for the BERT model (27), that the size of the fine-tuning dataset can impact performance drastically (66). Moreover, if the data is biased, it can result in poor generalization and hinder the model's ability to learn the intricacies of the DSL effectively (103; 102). This, in turn, necessitates an effective method for the generation of training data, keeping in mind that the quality and quantity of the fine-tuning dataset are paramount for achieving success.

In this study we aim to create an LLM that, given a prompt containing what type of chemistry simulation is needed, is able to create the necessary ORCA input file to run the requested chemistry simulation. In doing so, we explore different ways of generating a supervised dataset of ORCA input files. Using this data, we experiment with and evaluate the results of finetuning. Moreover, we compare different prompt-engineering techniques, and use RAG to determine whether these techniques can aid an LLM in ORCA input file generation. Finally, we determine whether applying these techniques allows our model to outperform the current state-of-the-art. We strive to answer the following research question:

Research Question. *How do prompt-engineering techniques, retrieval-augmented generation, and fine-tuning impact the ability of a large language model to generate accurate and functional input files for ORCA chemistry simulations?*

In exploring the answer to this question, our study offers a comprehensive case study of ORCA, exploring various approaches for creating a specialized LLM tailored to a DSL. Section 2 provides further insight into related studies. Section 3.2 describes the used LLM and extensively explain how and what techniques were used to optimize the LLM's performance. Furthermore, Section 3.1 describes how we created the datasets for used finetuning and Section 3.3 goes into the setup of the performed experiments. Section 4 shows the results for the experiments that were run and Section 5.1 discusses them. Finally, Section 5.2 mentions possibilities for follow-up research based on our findings.

2 Related Work

This section provides an overview of state-of-the-art LLMs, including those specifically trained for code synthesis, and how they have been applied in chemistry,

2.1 Large Language Models

There have been various successful approaches to building LLMs in recent years, nearly all of which built on the Transformer architecture (107). This architecture's self-attention mechanism enables the model to comprehend long-range context and dependencies within text, facilitating accurate predictions and coherent text generation. Its parallel computation also supports efficient training on large datasets, which is crucial for scaling up LLMs and improving their language understanding and generation abilities.

BERT, for instance, is a bidirectional variant of the Transformer architecture which has shown state-of-the-art performance in a variety of NLP tasks (27). GPT-3 (12) (175 B parameters) and GPT-4 OpenAI (75) (OpenAI did not reveal either the weights or the technical details of GPT-4) are unidirectional models that are also based on the Transformer architecture. However, whereas BERT focuses on downstream finetuning, GPT focuses on few-shot learning. Countless variations of BERT and GPT have been proposed in recent years (1; 54). InstructGPT (76) uses reinforcement learning to fine-tune the model with human feedback and, thereby, reduce toxic output (40). A different architecture named Pathways Language Model (PaLM) (21) is a model by Google with 540 B parameters that takes advantage of the

Pathways system to efficiently train on enormous amounts of data. Gemini 1.5 (100) was more recently released by Google and one of its most notable features is the extended context window of 128,000 tokens. This allows the model to consider a much larger chunk of text preceding a user-prompt, enabling it to better grasp the context and relationships between text. Anthropic developed the Claude family of models (33), with the most recent version being Claude 3. Claude 3 excels at processing and integrating different data types and is able of self-improvement through the use of Dynamic Neural Architecture Search (131). Lastly, we also highlight the current state-of-the-art open-source model: LLama 3 by Meta (104). It focuses on efficiency while maintaining performance and comes in various sizes, with options ranging from 8 billion to 70 billion parameters. For a more extensive overview of large language models, we refer the reader to (67).

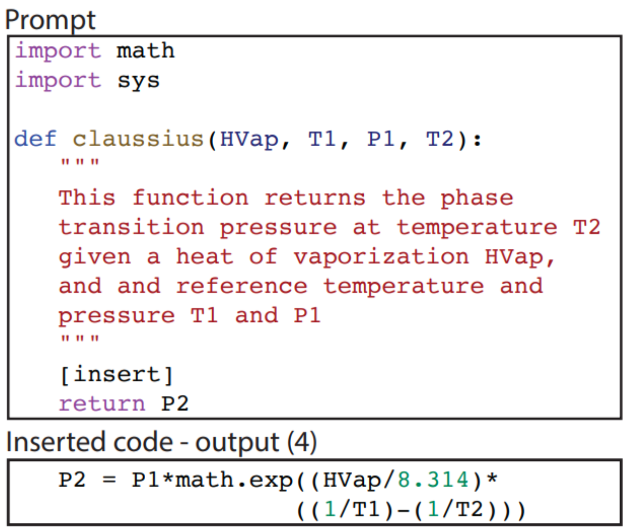
The above models are able of code synthesis, but not specialized on the task. In the past, specialized code synthesis models were encoder-decoders that made use of Abstract Syntax Trees (72; 114; 26; 42). An AST is a hierarchical, tree-like representation of the syntactic structure of source code, where each node represents a construct occurring in the source. It captures the structure of the code and how different elements are connected. More recently, LLMs have seen a rise in use in the context of code synthesis. Most LLMs in that regard are causal decoding-only models (121), meaning that they have to be prompted with a piece of text to be able to propose a continuation of said text. Codex (17) is a GPT-based model, which was fine-tuned on Python code from Github to generate code based on Natural Language (NL) comments and function naming. CODEBert (36) is a variant of the BERT model that is trained on six Programming Languages (PLs) to capture the semantic connection between the NL and PL. It is trained on both bimodal data (NL - PL pairs) and unimodal data (PL - PL pairs). PYMt-5 (22) is an extension of the T5 model (84) that jointly models Python source code with NL docstrings. Docstrings are descriptions or documentation texts that are embedded within code to explain its purpose, usage, and functionalities. Despite the advancements that LLMs bring to code synthesis, challenges remain, such as generating code that superficially appears correct but fails to perform the intended task, struggling with parsing long and complex specifications, and invoking elements outside the scope of the codebase (17).

2.2 LLM Applications to Chemistry

There have been various studies on how LLMs can be used in the context of chemistry. One example task that they are used for is molecule editing. More specifically, modifying a part of the molecule such that the resulting structure optimizes a desired property while making sure that other properties remain similar. Liu et al. (62) adopted a multi-modal approach, creating the MoleculeSTM model, which simultaneously learns from both the chemical structures and textual descriptions of molecules to assist in this task.

LLMs are also widely used in research on molecular property prediction. SMILES-Bert (112), `ChemBERTa` (19), and `ChemBERTa-2` (2) are instances of LLMs that were both built to map the SMILES (Simplified Molecular Input Line Entry System) notation (117), which is used to represent chemical structures with ASCII strings, to natural language and vice versa. By using this model, molecular properties could be predicted from a given SMILES notation.

In contrast, code synthesis for chemistry simulations has been relatively unexplored. The most relevant previous study to our work is most likely the one conducted by Hocky and White (44). In their research, they utilized the Codex model (17) to generate input files for the Gaussian software package.


Prompt

```
import math
import sys

def claussius(HVap, T1, P1, T2):
    """
    This function returns the phase
    transition pressure at temperature T2
    given a heat of vaporization HVap,
    and and reference temperature and
    pressure T1 and P1
    """
    [insert]
    return P2
```

Inserted code - output (4)

```
    P2 = P1*math.exp((HVap/8.314)*
                     ((1/T1)-(1/T2)))
```

Figure 2: An example prompt and the corresponding completion provided by `code-davinci-002` for a chemistry-related problem. Note that the model only completed this problem correctly three out of five times.

However, their approach involved the generation of Python scripts, which in turn were used to create Gaussian input files, rather than directly producing the Gaussian input files themselves. In a small explorative experiment, they report three trials of Codex generating code to create a Gaussian input file for a single point calculation based on a supplied comment. In doing so, they prompted the model to use `rdkit` library to get proper coordinates for the molecules. They found that the generated code rarely had syntax mistakes, showed that the model had some basic chemistry knowledge but often failed in obvious ways like not importing a necessary library or expecting a different data type to be returned by a function. Their study differs from our work in that we are tailoring an LLM directly to generate code for the DSL in question. Moreover the study focused at exploring the ability of Codex to help chemists in their research and in doing so ran the model limited times with some basic prompting. Our study differs from this in that we provide a full case study on building the optimal system with the aim of getting to improved performance over the current state-of-the-art with respect to ORCA input file synthesis.

While there is next to no research on DSL-synthesis like ours, there are various examples of LLMs being used for code synthesis in chemistry for general purpose programming languages. (119) assessed the general chemistry knowledge of various LLMs like Codex and the `GPT-3.5` variants `code-davinci-002` and `text-davinci-003`. They had the models solve chemistry problems posed as coding tasks across different domains in chemistry and chemical engineering. The authors created these problems by making use of their expertise in the field. Their research demonstrated that LLMs possess the capability to generate accurate code across diverse topics in computational chemistry, including the creation of chemical simulations. An example problem they provide is shown in Figure 2. Additionally, they offered valuable prompt engineering strategies for future investigations. These strategies include pre-importing relevant libraries before requesting code completion and incorporating copyright notices at the beginning of prompts, as to encourage the model to generate code with a higher degree of professionalism.

Boiko et al. (10) proposed an advanced model called `Coscientist`. Given a user prompt, a planner module based on GPT-4 determines whether and how to use either of the following modules:

- Web Searcher: Transform the user prompt into queries for the Google Search API or browse the web and report relevant information back to the planner.
- Documentation Searcher: An LLM is used to retrieve and summarize relevant documentation for commands necessary to use the hardware.
- Automation: A module is used to send commands directly to the hardware that is required to perform an experiment.
- Code Execution: Perform code execution using an isolated Docker container.

In this model, the planner is the model used for code synthesis. It uses the described modules to determine and write the Python code necessary to perform the chemistry experiment. Moreover, it is used to change the written code to handle software errors in case they appear. As a proof of concept, they showed that the model was able to write working code for six tasks, including controlling a robotic liquid handler and performing Suzuki and Sonogashira cross-coupling reactions. Bran et al. (11) also proposed a sophisticated LLM chemistry agent called `ChemCrow` that is able to make use of 18 tools ranging from general ones like web search to molecule tools that allow the agent to convert molecule names to their SMILES representations and safety tools to check whether a molecule is explosive or not. `ChemCrow` uses a ReAct (125) based workflow to reason about what tools to use given a user prompt. In short, this entails using a chain-of-thought reasoning (116) process with basic action plan generation (106; 120; 43).

2.3 Synthetic Data Generation in Code Synthesis

Synthetic Data Generation (SDG) is the process of creating artificial data that mimics the features, structures, and statistical attributes of data from the desired distribution. It encompasses a wide range of methodologies, like data augmentation (68), making use of generative models (32) and statistical models like Gaussian Mixture Modelling (20).

Whereas in our study we are creating a dataset from scratch, data augmentation is about creating new data by applying transformations to existing data. Given a dataset, it allows for increasing its volume, quality and diversity. It has been widely used, also in natural language processing (35), but is considered more challenging in code synthesis because random changes to code can strongly affect the semantics or quickly introduce syntax errors. Yu et al. (128) created a semantic-and-naturalness preserving auto-transformation tool (SPAT) that made use of 18 transformation rules to augment Java code. One example of this is replacing a 'while' statement with a semantic-equivalent 'for' statement. They evaluated this framework on code clone detection, method name prediction, and showed improved model-performance using the larger augmented dataset compared to the initial training set. Another approach to augmenting code, different from this rule-based method, is by learning proper transformations from examples. Jiang et al. (47) took this approach and proposed a state-of-the-art transformation inference approach called GENPAT, which is able to use a code corpus to statistically model what changes can be made to a program. Rolim et al. (87) used a similar strategy called REFAZER, but adopted a more supervised way of learning by using code before and after edits as training data to learn new possible transformations.

Direct generation of data for code synthesis has long been receiving academic attention, with older heuristics like genetic algorithms (79) and graph models (45) seeing some success. More modern approaches tend to make use of generative models, however, only to model the underlying data distribution of a programming language and sample from it to generate new data. Yin and Neubig (127), for instance, proposed a model that specifically considers the syntax of the target programming language. An encoder-decoder model (LSTM as encoder, standard RNN as decoder) first generates an AST, which represents the structure of the code, and then converts the AST into actual code. They make use of a grammar model that defines the valid ways to generate an AST based on the programming language's syntax rules and showcase effective Python code generation. There are also various studies that focus on synthetic generation of DSL code. Parisotto et al. (80) used a Recursive-Reverse-Recursive Neural Network to encode and expand a partial program tree into a full program tree in a DSL for regular expression-based string transformations. Shin et al. (92) argue that this type of approach can result in poor generalization to unseen data and propose a method for creating DSL training datasets with a more uniform distribution. They achieve this by controlling and evaluating the bias of synthetic data distributions, defining salient random variables that capture desired features of the program and input spaces (such as the number of parentheses in a calculator expression) and specifically manipulating their distributions. They demonstrate an increase in cross-distribution test accuracy, albeit with a slight decrease in on-distribution test accuracy.

In our case, we are generating both ORCA input files and the input prompt of a user, which can be regarded as a description of the code. Alon et al. (4) proposed a model for generating code-descriptions with their `code2seq` model. It uses a sequence-to-sequence model that encodes code as ASTs. The decoder is trained, using attention mechanisms, to generate a description based on the encoded representation of the AST. They evaluate their model on Java code summarization and C# code captioning and show great generalisation performance across different programming languages. A very similar work by Karia et al. (49) changed the code2sec architecture slightly by incorporating a transformer into it. Allamanis et al. (3) used a bimodal model to perform both code captioning and source code retrieval given a caption. While they train the model on a synthetic dataset to test the ability of the model to learn, they never translate this model trained on synthetic data to real-world performance. They do, however, show that the model is capable of code captioning for two C# datasets, outperforming baseline models. Note that code captioning for DSLs, to our knowledge, has yet to be researched. Nevertheless, the above papers were able to create real-world datasets through gathering natural language questions and the corresponding code snippet from StackOverflow and Do Net Perls.

For further reading on SDG techniques, we refer the reader to Bauer et al. (6).

2.4 Prompt Engineering

As mentioned in Section 1, prompt engineering has become an essential way to make optimal use of the capabilities of LLMs, especially also in code synthesis. We make a distinction between mere prompt engineering techniques and full methodologies, and we highlight techniques used in this study and in code synthesis in general.

As mentioned previously, White et al. (119) found that LLMs were better able to code the solution to chemistry problems when relevant libraries were imported in the initial code snippet and when copyright notices were incorporated as comments at the top of the code to be completed. Similarly, OBrien et al.

(73) studied how the presence of TODO comments impacted the quality of GitHub Copilot's generated code, and found that its inclusion can actually cause the model to regenerate problematic code described in the TODO comments. These studies show how LLMs (which in essence are next-token-prediction models) can easily be manipulated with prompting techniques to target certain patterns they learned in their training data. Even including simple phrases like ”imagine you are” can guide the model to generate more contextually rich and creative responses (Koul). Wang et al. (110) showed in a study of CodeBERT that the order of tokens, even if they have the same semantics, can greatly influence model performance. Additionally, they showed that the prompt length is important as well, and that longer prompts do not always improve performance. Furthermore, Brown et al. (13) showed that using few-shot learning, which entails showing the model examples of the target task in its context window, can match state-of-the-art fine-tuned systems in various tasks such as question answering.

Aside from using similar prompting techniques to those described above, reasoning-based-prompt engineering is a large part of the prompt engineering part of our research. Reasoning-based-prompt engineering entails having a model first reason about the problem at hand, before generating a solution. One popular prompting-methodology in this space is Chain-of-Thoughts (CoT) (116). It has been demonstrated that when LLMs generate chain-of-thoughts— a series of intermediate reasoning steps— their performance in various tasks significantly improves (116; 16; 90). Due to its success, there have been various studies proposing extensions and variants of CoT (129), including ones aimed at code synthesis (88). Chain of Code (CoC) (58) tried to extend upon CoT by leveraging code-writing and having the LLM write pseudocode for different subtasks to make them 'think in code'. Similarly, Structured Chain of Thoughts (SCoT) (59) prompts the LLM to incorporate program structures into the reasoning steps and to have the LLM describe what part of the program should be used in what structure.

Other methodologies take a less linear approach when compared to CoT. Tree-of-Thoughts (124) has the LLM manage a tree-like structure of intermediate reasoning steps where different 'thoughts' are considered paths to a final solution. It allows for search algorithms to be used on the LLM's thought process and can allow for the LLM to evaluate different reasoning chains in getting to a final solution. Dainese et al. (23) used this approach to aid in generating Python code for model-based reinforcement learning. A similar approach is Graph-of-Thoughts (GoT) (9) prompting, which instead of sequentially modelling thoughts takes on a more non-linear approach. It tries to have the LLM formulate its reasoning as a directed graph, which is a structure that has in the past been shown to aid in program synthesis (8; 38). As LLMs have been shown to be especially prone to hallucinations in code synthesis (61; 101), prompt engineering techniques designed to mitigate this have also been explored. Our work investigates Chain of Verification (CoVe) (28), which prompts an LLM to have a second look at the responses they give and have them consider whether they made any mistakes. Ngassom et al. (70) managed to increase the amount of executable code generated by LLMs by 13% over the previous state-of-the art using CoVe. Through creating an AST out of the LLM's code, and asking several verification questions for each node in the AST, they try to correct potential mistakes in the generated code.

Note that tuning prompts to get to optimal results is a considerable task in itself. Liu et al. (63) proposed to continuously adjust prompts with a separate frozen LLM and measure whether performance increases for different phrasings or slightly adjusted content. Moreover, Shin et al. (93) took a different approach and made use of gradient-guided search.

Lastly we highlight the study by Ridnik et al. (86). They proposed a full code prompt engineering flow called `AlphaCodium`, where they make use of various prompting techniques to solve competitive programming problems. It breaks down the solving process of a coding problem into stages focused on understanding the problem, generating possible solutions, and then testing and refining those solutions. It uses various prompting techniques in getting to a final answer. When reasoning about problems, `AlphaCodium` is asked to use bullet points to encourage a deeper understanding and a more organized thought process. The model is constantly asked to review and refine its outputs by being prompted in a CoVe way. Multiple solutions are considered in a GoT type way, `AlphaCodium` ranks the possible solutions based on factors like correctness, simplicity, and efficiency. The research gives clear insight into how much a prompt engineering flow can improve LLM performance, as they show an increase in GPT-4 accuracy for the pass@5 dataset from 19% to 44% of the problems being solved.

For further reading on different prompt engineering techniques, of which there are many more, we refer the reader to Sahoo et al. (88).

2.5 Retrieval Augmented Generation

Retrieval Augmented Generation was initially proposed by Lewis et al. (57) to help improve LLM-performance in knowledge-intensive NLP-tasks by providing the model with non-parametric memory. The paper describes using a retriever (DPR (50)) to retrieve relevant external context and a generator (BART (108)) that takes this as input to produce a more accurate output. They also consider a distinction between using the same retrieved document for the entire generated sequence and choosing content from different retrieved documents for each word it generates. As in our study, the retriever is not directly trained on what document is to be retrieved.

RAG has also been widely used in code synthesis, among other tasks, as it has been shown to reduce hallucination (94). Wang et al. (113) used RAG as a means for automatic program repair through retrieving very small code 'patches' (which can be single lines) from a database of well-functioning code. Based on a given code snippet, they have the CodeT5 model (115) debug the initial code snippet. Parvez et al. (81) proposed the `REDCODER` framework, which utilizes GitHub and Stack Overflow databases to directly retrieve code snippets based on a given prompt. The framework then employs the LLM to adjust the retrieved code snippet according to the initial prompt, if necessary. The ReACC (64) is near identical to this, but solely focuses on code completion. ProCC (99) extends on the above studies by employing three prompt templating techniques to get outputs from the LLM that are used to retrieve different pieces of code from the documentation database. These techniques involve:

1. Having the LLM encode the lexical semantics of the prompted code.
2. Having the LLM propose a hypothetical line to add to the prompted code.
3. Having the LLM summarize the prompted code.

The process of choosing which retrieval perspective to use for code completion is handled by an Adaptive Retrieval Selection Algorithm. This algorithm employs a learning approach to dynamically select the most suitable perspective. The LinUCB algorithm, a variant of the multi-armed bandit algorithm (97), is used for its ability to continuously learn and adapt based on outcomes.

All these studies focused on general purpose languages, but in our case RAG supposedly has even more effect as it contains information the model has not seen. The paper by Baumann et al. (7) studied whether RAG (and few-shot learning) can help an LLM understand the syntax of a DSL it was not trained on. They show that RAG can be used to enable simple model synthesis for Monticore generated, and thus uncommon, DSLs, as long as there is a fitting knowledge base that can be accessed to provide the needed examples. Closest to our implementation of RAG is the study by Zhou et al. (132), which proposes a general documentation-centered approach called `DocPrompting` to solving code synthesis. Given an NL intent, the goal of `DocPrompting` is to generate a code snippet in a particular PL. The model accesses a collection of documentation to assist in this task based on a similarity score between the intent and the documents. The top-k documents with the highest similarity scores are retrieved and used to aid in code generation.

Note that Gao et al. (39) provide a comprehensive survey on RAG for further reading.

2.6 Finetuning

To our knowledge, there is no existing literature on fine-tuning an LLM specifically for a DSL. Therefore, we will limit this subsection to a brief overview of studies that demonstrate the effects of fine-tuning on target tasks with limited data and on fine-tuning for code synthesis.

Ogueji et al. (74) showed that finetuning BERT on less than 1 GB of only low-resource African languages allows the model to outperform and be competitive with various more extensively trained models across African text classification and entity recognition tasks. A study that extensively researches the effect of finetuning on LLM performance is Dodge et al. (29). They fine-tuned BERT models on multiple GLUE benchmark datasets, varying only the random seeds. They found significant performance variability due to different seeds, influenced by weight initialization and training data order. On smaller datasets, many runs diverged during training, and they recommended early stopping for less promising runs. In our work, we use LLMs to generate the input for the LLM. Huang et al. (46) did the opposite and used a pre-trained LLM to generate labels for unlabeled questions using CoT prompting and CoVe. They then used this generated data to fine-tune that same LLM. Their results are promising and show state-of-the-art-level reasoning performance on various benchmarks.

Austin et al. (5) measured the influence of few-shot-prompting and finetuning on a wide collection of models with increasing parameter size to solve python coding problems. Their fine-tuning was quite small and consisted of 374 synthesis problems. Nevertheless, they showed that fine-tuning on a held-out portion of the dataset improved performance by about 10 percentage points across most model sizes. Furthermore, Dong et al. (30) also explored finetuning for general code synthesis using the Code Alpaca codebase (14). Their most relevant finding to our study was that, in solving python problems with the various finetuned models, code generation consistently improved with increasing amount of data.

For further explanation on finetuning, regularly used methods to build finetuning datasets, and popular finetuned models we refer the reader to the paper by Zhang et al. (130).

2.7 Program Feedback

ORCA provides users with extensive output files from chemistry simulations, along with error messages when the simulation did not run correctly. While our study uses this output only in a limited way, it is crucial for our dataset creation. Consequently, we briefly highlight studies that have utilized program output in similar contexts.

Closest to our context is the paper by Skreta et al. (96), where they used GPT-3 to generate a DSL for executing chemistry experiments with a robot. To ensure the validity of the generated code, they employed a separate model as a verifier. This verifier functions as both a syntax checker and a static analyzer, which checks the output of the language model and provides feedback. They use this in a feedback-loop with the LLM to keep improving the code until it works or the verifier has been used a pre-specified amount of times. Moreover, like in our study, they analyze the frequency of specific types of errors and they found that LLMs most often produced a wrong parameter. A similar approach was explored in (71), but it was more general-purpose in the sense that it proposed a framework that could be applied to any PL. In this paper, the so called `LEVER` introduces a learning framework that leverages execution results to verify the correctness of generated programs. It differs from (96) in that the verifier is trained to distinguish between correct and incorrect programs based on the joint representation of program output, natural language description, and program surface form. The verifier assigns a verification probability to each generated program, indicating the likelihood of the program being correct. Additionally, instead of using the verifier in a feedback loop together with the LLM, the aggregated probability score is used for reranking the multiple LLM-programs, and the most probable programs that execute to provide the correct result are outputted. They show improved performance over baseline coding-LLM's across both SQL and Python coding datasets. Peng et al. (82) proposes a general framework called `LLMAugmenter`. This framework iteratively enhances LLM-generated prompts to improve the coherence and grounding of the responses using feedback derived from various utility functions and external knowledge. The primary goal is to generate responses that are contextually coherent and well-grounded in consolidated evidence. For instance, to evaluate how well the generated responses are grounded in evidence, they employ the Knowledge-F1 score (95) and have the LLM model be self critical if this is low.

3 Methods

3.1 Dataset Creation

A major part of our research involves fine-tuning (Section **??**). This process allows our LLM (Section **??**) to learn the structure of ORCA input files, which it likely has not encountered sufficiently during pretraining. This fine-tuning should presumably enable the model to perform the task of ORCA code synthesis more effectively.

To facilitate fine-tuning, we need pairs of natural language prompts and ORCA input files to teach the model the code synthesis task. However, due to the nature of ORCA as a domain-specific language (DSL), there is limited data available for further training. Additionally, to accurately assess real-world performance, it is crucial to use real-world data for our test and validation sets which makes for that the little data available has to be used for evaluation. Therefore, we propose and employ multiple ways to generate supervised ORCA input files and a method to synthetically create a user prompt from a given input file. This section will go over and explain these methods.

Note that we decided to first generate ORCA input-files and from these input files generate the user-prompts due to the efficiency of this order. In generating our data, we focused on generating runnable ORCA input-files, so that we know they have some resemblance to real world data. If we would first create user-prompts and then create corresponding ORCA input files based on those, we would still have to check if these are runnable and if they are not, the user-prompt ends up being invalid as well. Doing this in the reverse order eliminates the extra step of possibly generating an invalid user-prompt.

3.1.1 Input File Generation

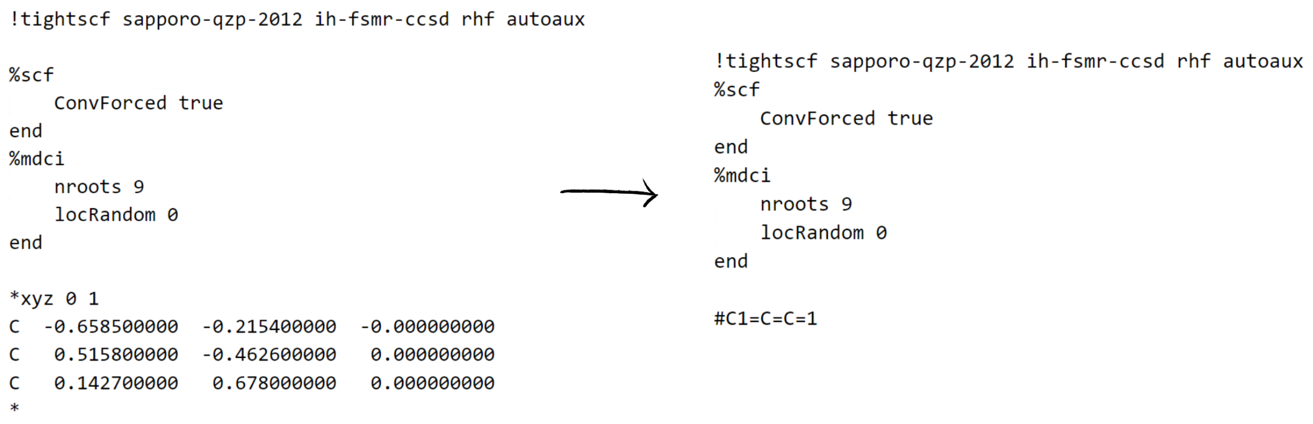
For generating an ORCA input file (Figure 1), we have to consider three types of code to generate: keyword-lines, input-blocks and a coordinate-block.

Keywords are, like mentioned in Section 1, specific terms that direct ORCA to perform particular tasks. Examples are energy calculations, geometry optimizations, frequency analyses but they can also specify computational methods, basis sets, and convergence criteria. The order of keywords does not matter for the functionality of ORCA, nor does it matter if they are split over multiple keyword lines or if they are capitalized or not.

Nearly all these keywords can also be defined through more detailed input blocks, which provide greater control and customization for advanced users. They allow for various functionality that is inaccessible through keywords, such as changing the parameters for density functionals, defining custom basis sets and grids and loading settings or coordinates from external files. For every option (like scf in Figure 1) various settings are defined, which in turn all have their own range of valid values. Note though, that when one can choose between keywords or an identical definition in terms of input blocks, keywords are generally preferred for their simplicity, ease of use, and readability. They facilitate quick setup, minimize errors and promote standardization. Therefore we take a keyword-centric approach in our generation methods.

The coordinate-block is essential for ORCA calculations because it defines the precise spatial arrangement of atoms in the molecule, which directly influences the electronic structure and properties. Accurate coordinates are necessary for solving the Schrödinger equation, determining interatomic interactions, and applying basis sets correctly. This ensures that the predicted molecular behavior, energy, and reactivity are reliable. Without correct coordinates, the results of the quantum mechanical calculations would be inaccurate, leading to erroneous conclusions about the molecule's properties. For all three generation-methods, we employ the same method of generating the coordinate block. Namely, we simply append a valid coordinate section of a randomly chosen molecule. These molecules are chosen from a generated dataset containing valid molecules of up to three atoms. The appended molecules are randomly selected from a dataset containing valid molecules of up to three atoms, generated using the `rdkit` library to ensure validity of both SMILES representation and coordinates. We opted to generate molecules with a maximum of three atoms. Smaller molecules significantly expedite ORCA calculations, and three-atom molecules are the smallest that allow for the full range of ORCA calculations. The molecules we generated may include a radical electron to support the unrestricted Hartree-Fock method but exclude charged atoms or uncommon or heavy atoms (B, P, S, Br, and I).Namely, we randomly choose a molecule from a generated dataset of molecules to append to the generated keyword line(s) and input block(s).

However, it is difficult for an LLM to generate such a code block, due to the fact that it is mainly made up of of floats and not NL-like code. These floats also have an organized structure. Moreover, like we show in generating our molecule dataset, one can use tools such as `rdkit` (56) that can generate a valid coordinate structure for a given molecule. Therefore, we decided that teaching our LLM to generate accurate coordinates would be a waste of resources and unnecessarily complicate the task. Instead, our focus is on teaching the model to associate appropriate basis sets with specific elements and molecules.



```
!tightscf sapporo-qzp-2012 ih-fsmr-ccsd rhf autoaux

%scf
    ConvForced true
end
%mdci
    nroots 9
    locRandom 0
end

*xyz 0 1
C  -0.658500000  -0.215400000  -0.000000000
C   0.515800000  -0.462600000   0.000000000
C   0.142700000   0.678000000   0.000000000
*
```

```
!tightscf sapporo-qzp-2012 ih-fsmr-ccsd rhf autoaux
%scf
    ConvForced true
end
%mdci
    nroots 9
    locRandom 0
end

#C1=C=C=1
```

Figure 3: The transformation of a coordinate block (left) to a SMILES molecule (right).

To achieve this, we replace the coordinate block for valid input files with a comment containing the SMILES representation of the molecule, as shown in Figure 3.

Considering the above, we propose three methods for generating ORCA input files:

1. **Brute-force method:** This method does not consider any interactions between keywords and input blocks, and instead combines documentation randomly.
2. **Manual-based generation:** This method relies on the input files provided in the ORCA manual (version 5.0.4). It combines full sections from different input files instead of the smaller parts of sections. The assumption here is that by extracting full individual sections, we have the sections represent accurate combinations of keywords, options, and settings.
3. **Rule-based generation:** This method generates input files based on a specific ORCA calculation type, using predefined rules to ensure the creation of coherent and accurate input files. It uses prior information on the ORCA software, to be able to create a system that is able to consistently generate accurate input files

The following sections will provide all details of these three input-file generation methods.

3.1.1.1 Brute-force

The most straightforward way to generate an ORCA input-file is to randomly combine the parts of an ORCA input file. This is exactly what we do in the brute-force approach. Namely, we randomly combine keywords, options and settings in hopes of getting to a valid input file. To get access to these, we scraped all options and keywords out of the ORCA manual (these are readily available in chapter 6). Moreover, for the settings we scraped all input-blocks shown in the ORCA manual and out of these extracted all settings as a full line (both setting-option and setting-value). Note that we kept track of what setting-line belonged to what option. This extracted documentation was then used as follows:

1. For the **keyword-line**, we select a sample of up to ten keywords out of all keywords defined in the ORCA manual.

2. We randomly choose if an input-block is added to the input file. If it is, we randomly select an option, and then randomly choose up to five settings that are defined for the selected option.

3. We randomly choose molecule out of our dataset, and append a corresponding coordinate section to.

4. We try to run the file. If it succeeds, we transform the coordinates to SMILES and save it to the dataset.

Presumably, this method can be quite slow in getting to generate valid ORCA input files, as combining elements without using any prior knowledge on if they work together could result in many invalid input files being generated. Moreover, even if they are runnable, there is a chance of them not reflecting how a real-world input file looks. It is however a very easy to implement way of generating the input files, that is also very generalisable to other DSLs.

3.1.1.2 Manual-based

3.1.1.3 Rule-based

3.1.2 User-Prompt Generation

3.2 Model Architecture

A full overview of our model architecture is shown in Figure 4. This section will go on to explain the details of this architecture and in doing so, explain our choice of LLM, our use of prompt engineering and how we employ finetuning and RAG. Note that the definition of these concepts was provided in Section 1 and that we thus therefore refrain from reintroducing them in this section.

3.2.1 Inference

Given an system prompt (S), an input-prompt (X), we have our LLM (m) produce output o. This is put more formally in Equation 1 and is also denoted in Figure 4.

$$o = m(S, X) \tag{1}$$

The system prompt consists of depends on the chosen prompt engineering technique, of which the implemented ones are described in Section 3.2.3. Moreover, the standard templated user-prompt (Section 3.1.2) can be extended with external context if RAG is used (Section 3.2.5).

Presumably, o contains an ORCA input file $\hat{y}$, which we try to extract in two ways. The first being by through checking of the LLM produces a code block, and if so extracting what is in this block. The second is with a regex search based on the structure of an ORCA input file. If both of these fail, we use o as $\hat{y}$.

The goal of this task is to have $\hat{y}$ be as similar as possible to the true ORCA input file y. Similarity is measured using the metrics described in Section 3.3.3. Note that, as previously described in Section 3.1.1, these ORCA input files do not contain the coordinates of the used atoms, as we decided that this information was not ideal for an LLM to learn.

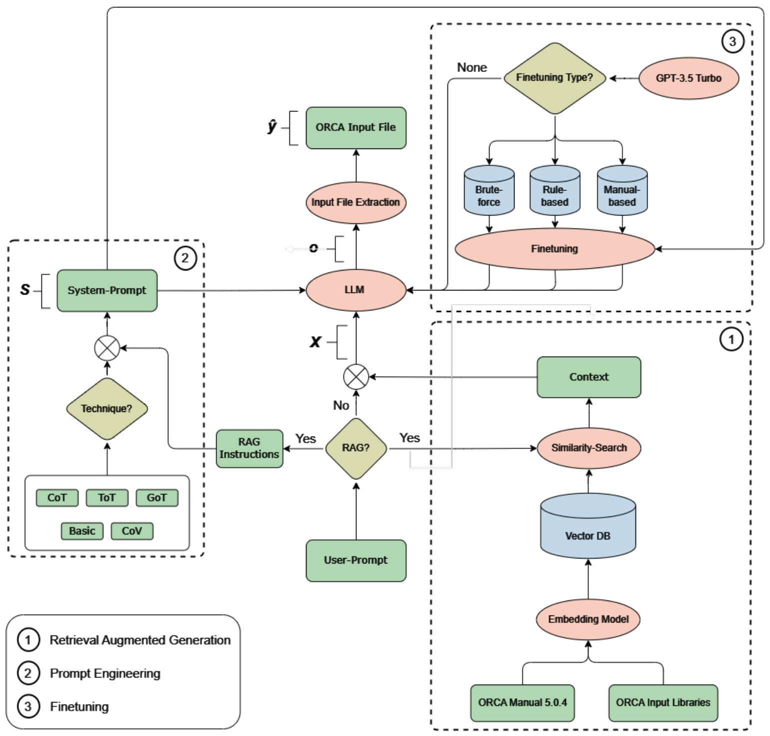


Figure 4: A full overview of the used model architecture.

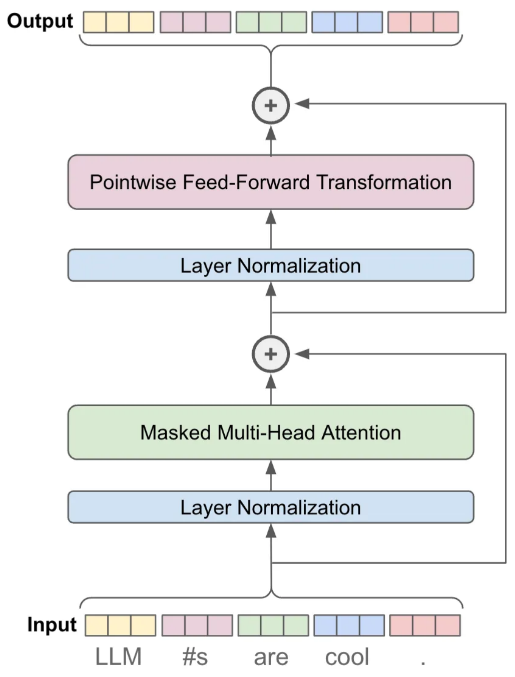
3.2.2 GPT-3.5 Turbo

The LLM is central to our methodology as it is the base we are building from to create a specialized model. Given a user-prompt containing the description of a desired chemistry simulation, the LLM is supposed to generate an ORCA input file that corresponds to this description. Moreover, it is the base model which gets fine-tuned (Section 3.2.4) on the synthetic datasets (Section 3.1) and the model that has to interpret and use context retrieved by RAG (Section 3.2.5). The model we use in this study is `gpt-3.5-turbo-0125`.

GPT-3.5 Turbo is an updated version of GPT-3 (12) that uses different training paradigms to get to improved performance. Note that the full technical details of its training process are not made public. We do know that it was pretrained, like GPT-3, on a blend of text and code published prior to September 2021 and that its architecture is presumed to be nearly identical. Therefore, we will provide the reader with a short description of this architecture. The GPT model-family are all based on the transformer architecture and uses multiple instances of the Transformer decoder block (shown in Figure 5).

A single Transformer decoder block comprises several components:

- **Input Embedding:** The raw input tokens are first converted into dense vectors of fixed size, known as embeddings. Each token is mapped to a high-dimensional space.
- **Positional Encoding:** Since the Transformer model is not inherently sequential, positional encodings are added to the input embeddings to provide information about the position of each token in the sequence.



Output
Pointwise Feed-Forward Transformation
Layer Normalization
Masked Multi-Head Attention
Layer Normalization
Input
LLM
#s
are
cool
.

Figure 5: The Transformer decoder block.

- **Multi-Head Self-Attention:** This mechanism allows the model to focus on different parts of the input sequence when processing each token. It computes attention scores using query, key, and value vectors derived from the input embeddings. These scores determine how much focus should be placed on other tokens when representing a given token. Multiple attention heads allow the model to capture various aspects of the sequence.

- **Feedforward Neural Network:** After the attention mechanism, the output is passed through a feedforward neural network, which consists of two linear layers with a ReLU activation in between. This helps in transforming the attended features into a more refined representation.

- **Layer Normalization and Residual Connections:** To stabilize and improve the training process, layer normalization is applied after each sub-layer (attention and feedforward), and residual connections are used to ensure gradients flow smoothly through the network.

To train these decoder blocks, and thus GPT-3, a next-word prediction task was used. This is a form of unsupervised training where the model predicts the next word in a sentence. It involves presenting the model with input sequences and predicting the next token in an autoregressive manner. The model starts with random parameters, and through iterative updates based on prediction errors, it learns to generate coherent text. The training objective is to minimize the difference between the predicted tokens and the actual tokens in the training data. To tokenize sentences, it uses a subword tokenization algorithm called byte-level Byte Pair Encoding (91). This algorithm breaks down words into smaller subwords, allowing the model to handle rare or unseen words by representing them as a combination of known subwords.

In our study, we use the most advanced variant of the GPT-3.5 versions: `gpt-3.5-turbo-0125`. It offers higher accuracy in responding to requested formats compared to other versions. It has a context window of 16,385 tokens and is able to return a maximum of 4096 tokens. The base of `gpt-3.5-turbo` is the `code-davinci-002` model, which is trained for code generation. OpenAI then used supervised fine-tuning to create `text-davinci-002`. To improve this model further, they used the Reinforcement Learning from Human Feedback (RLHF) training strategy (51) to create `text-davinci-003`. This method involves training a reinforcement learning reward model that gets used to evaluate the quality of responses generated by the model. The reward model assigns scores to different responses based on their relevance and quality. The model is then finetuned with Proximal Policy Optimization (89), which adjusts the model's parameters to maximize scores received by the trained reward model. RLHF showed improvement in GPT's its ability to understand instructions and generate text. This model is further optimized for chat and then becomes what we know as `gpt-3.5-turbo`. Note though `gpt-3.5-turbo` and `code-davinci-002` have been shown to produce very similar results (126).

We opted for using the `gpt-3.5-turbo-0125` model as it is, as of writing, the most widely used LLM due to its incorporation in ChatGPT. It is also among the state-of-the-art LLMs that can be finetuned (18). We opted to use a general-purpose LLM, and not specialized coding model like `CodeT5`, as we figured that these models have more knowledge of chemistry simulations due to their broader knowledge base and would thus be able to understand the user's prompt, especially if we tailor them to the task of ORCA input files. Moreover, this knowledge is necessary as we aim to provide the user of our specialized model to be able to use the LLM's chat-like interface to offer them a way to get to know more about the simulation in question or troubleshoot possible issues. This will be further discussed in Section 5. Also note that for the various more recently developed general-purpose LLMs finetuning is not available yet or is in an experimental phase and would thus not allow us to compare their performance with and without finetuning. This eliminated various models from our choice of LLM, like `GPT-4(o)` , `Gemini 1.5` (100), `Claude 3` (33), `DeepseekV2` (24) and `LLama 3` (104).

3.2.3 Prompt Engineering

To optimally make use of `gpt-3.5-turbo`, we employ various prompting techniques. Note that we employ these prompting techniques in the so-called system-prompt of the LLM, as is also shown in (2) of Figure 4. The system-prompt is the set of instructions or guidelines provided to the LLM that are used to define its behavior, tone, and the kind of responses it should generate. It can't be overridden and is used in every response of the LLM. This is different from the user prompt, which is the input or question given by the user to the LLM, in our case a description of a desired ORCA chemistry simulation. These user-prompts are not prompt-engineered and the way they are created is described in Section 3.1.

To make the results as comparable as possible, the system-prompts all have a similar structure in what information is to be reasoned about and verified but differ in their approach of having the LLM reason. As described in Section **??**, an ORCA input file consists of keywords and input blocks. Keywords can in turn be divided in various categories, most important of which are basis sets and density functionals. Input blocks have an option-type for which the settings in the input block are applied. The prompts are aimed at ensuring that the LLM reasons step-by-step through the construction of the ORCA input file.

Specifically, the prompts guide the LLM through the following:

1. Identifying the relevant ORCA keywords.

2. Determining the necessary input block options derived from the user description and the chosen

keywords.

3. Establishing the appropriate ORCA input block settings required to achieve the desired computational setup.

4. Identifying the SMILES notation of the molecule to be used within these keywords and input blocks.

5. Compiling the final ORCA input file by synthesizing the information from the previous steps.

We created six system-prompts, largely based on the literature described in Section 2, that aim to teach the model how to generate ORCA input files. The full prompts can be found in Appendix A, but we will briefly go over the methodologies here once more. Note that the **Basic** prompt serves as the basis for the other, more elaborate prompts.

The exact prompts are as follows:

- **Basic**: Here, we make use of various proven prompting techniques but do not use a central prompt engineering methodology. We instruct the model to behave as a chemistry expert, describe the format of an ORCA input file, and employ few-shot learning by showing it five examples of prompt and input file pairs (13). This choice of the number of examples is arbitrary but was made as keeping into account that including too many examples can lead to a decrease in performance (110).

- **CoT (Chain-of-Thought)** (116): This prompt defines five linear steps to gather the information described above. The model is instructed to write down the answer to all these steps, clearly separating and learning which part of the user prompt refers to which part of the input file.

- **CoVe (Chain-of-Verification)** (28): We first have the model make an initial guess of the correct input file. After this, we perform the same five steps as in CoT. However, instead of reasoning what should be used, we ask the model to verify if the initial choices are correct or if something needs to be adjusted.

- **GoT (Graph-of-Thought)** (9): This prompt instructs the LLM to use a non-linear way of mapping out its reasoning. We encourage it to find the core elements of the user prompt and to create connections between these elements. For all elements, multiple options should be considered to ultimately consolidate the most suitable options.

- **ToT (Tree-of-Thought)** (124): In this technique, we employ the model to explore different branches of thought and to consolidate towards a final, best answer. In the first step, we have the model write down multiple ideas for what the simulation could be.

3.2.4 Finetuning

To expose our model to more ORCA-specific-data and specialise our model on the task of ORCA input file synthesis, we finetune our model on the three datasets described in 3.1. A simplified overview of the selection of a finetuned model is also provided in (3) of Figure 4.

Formally, finetuning GPT-3.5-turbo involves adapting the model parameters θ to perform well on the supervised learning task of producing ORCA input files. Ideally, we minimize the objective function shown in Equation 2 while also maintaining the initial pre-trained performance of the LLM on other tasks, where $\mathcal{D}$ represents the training dataset consisting of N labeled examples $\{(x_i, y_i)\}$, where each example x_i is a user-prompt and y_i is its ORCA input file and S represents the system prompt used during finetuning. Note that this Equation holds for our case where we limited the supervised learning task to only the initial user-prompt and the ORCA input file which should be in the response, but that it can be extended to finetune on full conversations.

$$\theta^* = argmin_\theta \quad \mathbf{E}_{(\mathbf{S},\mathbf{x},\mathbf{y})} \sim \mathcal{D}[\mathcal{L}(\theta; x, y)] \tag{2}$$

To achieve this, the parameters θ are updated iteratively using a gradient descent algorithm. Unfortunately, the exact details of what gradient descent algorithm OpenAI uses is undisclosed, as is the loss function. We provide a general example of how an iterative update can look in Equation 3, where $\nabla_\theta \mathcal{L}(\theta_t; S, x_i, y_i)$ is the gradient of the loss function $\mathcal{L}$ with respect to the parameters θ evaluated on the example (x_i, y_i) using system-prompt S and η represents the learning rate.

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(\theta_t; S, x_i, y_i) \tag{3}$$

During fine-tuning, we use the same system-prompt across all examples so that the model learns to generate responses consistently and in alignment with the desired style, tone, and content specified by the system-prompt. In the case of running the model with **Basic** prompt-engineering we define the response, or label in the finetuning task, as only the desired ORCA input file. However, when using prompt techniques, it is necessary to create synthetic responses as we are trying to teach the model to reason in its output. We did this by altering y to start with the steps that are defined in all our prompts (Appendix A) and synthetically answering them based on the correct input file. An example of such a synthetic response for the CoT prompting methodology is given below.

```
I used the following step-by-step logic to get to my input file
    prediction:
Step 1: The basis sets used are dkh-def2-tzvpp.
No density functional is used. Further keywords used are:
    verytightscf,dkh
Step 2: The input block options that should be used are: casscf
Step 3: For casscf, the settings are as follows:
nel 2
norb 2
Step 4: The desired SMILES is B1=C=N1
Step 5: Given the answers to the above steps, the final ORCA input
    file is:
!dkh-def2-tzvpp verytightscf dkh
%casscf nel 2
norb 2
end
#B1=C=N1
```

3.2.5 Retrieval Augmented Generation

In this study, we use a naive-RAG scheme, which is the most basic RAG methodology (39). Due to its simplicity and explainability, it was deemed an appropriate initial approach for this research. An overview of our implementation is shown in (1) of Figure 3.2.

Our version of RAG uses a database of two documents that are presumed to be useful context for generating input files: the ORCA manual (version 5.0.4) and the ORCA input libraries website. The ORCA manual is the main documentation for the ORCA software. It provides extensive and authoritative information on the functionalities, commands, and parameters available within ORCA. We used the all pages of the document except the parts of chapter 6 where documentation definitions were provided. This decision was made because our user prompts were constructed using the natural language found in these tables. If we were to include these tables, RAG would likely retrieve these tables consistently, leading to an artificial inflation of the model's performance. This is because the model would be unfairly advantaged by the direct overlap between the prompts and the documentation definitions. By excluding these tables, we ensure a more accurate assessment of the model's capabilities, preventing it from relying on verbatim matches and instead encouraging genuine comprehension and generation abilities based on a broader context.

Moreover, ORCA input libraries was used as it offers a more comprehensive overview of the possible calculations in ORCA. It contains many practical examples and templates of input files, which can guide users in creating their own files. To be able to use it as a document, we scraped all pages of the website and saved each page to a page in pdf format.

Both documents were split by page, and each page was embedded using the `text-embedding-3-large` model, which is the state-of-the-art embedding model by OpenAI at the time of writing. The resulting embeddings were then indexed using FAISS (31), a library designed to efficiently cluster dense vectors. This allows for efficient retrieval through similarity search, ensuring that relevant information can be quickly and accurately located.

Given a user-prompt, we use FAISS similarity search between the prompt and all indexed embedding vectors. The *k* most similar pages are retrieved and added to the user-prompt so that the model can make use of it during the prediction of an ORCA input file.

During initial experimentation with RAG we found that the model did not indicate how it used context, often forgot to use context or use context when it was not applicable to the situation. We therefore opted to dynamically add extra instructions to the system-prompt whenever RAG was used. After quickly introducing what type of data we are using to retrieve the external context from, we have the model reason about the context. We prompt it to write down what part of the context it uses for what part of the input file and why. Moreover, we instructed the model be mindful to only use the context when it is actually helpful and relates to the users request and not use context if it is not useful. The full prompt can be found in Appendix A.

3.3 Experimental Setup

3.3.1 Experiments

The goal of the experiments was to explore the answer to the question of how we could create an LLM proficient at generating accurate and functional input files for ORCA, the chemistry simulation DSL. We split this into the following sub-questions, with performance referring to the performance of GPT-3.5-turbo in generating ORCA input files:

1. How does finetuning affect performance?
   (a) How does the amount of generated data affect finetuning performance?
   (b) How do the brute-force, manual-based and rule-based data-generation methods affect the performance of GPT-3.5-turbo in generating ORCA input files?
2. How does prompt-engineering affect performance with and without synthethic finetuning?
3. How does RAG affect performance?

To answer these questions, a range of experiments were performed where RAG, prompt-engineering and fine-tuning get combined in different configurations:

•

For every prompt-engineering-technique we use, the amount of models we get is increased by the amount of base models using the technique, added by the amount of models across the different finetuning models. This comes down to $3 + (3 * 2) = 9$, as there are three different finetuning datasets, where for each one two models need to be run using the prompt-engineering-technique. Due to financial limitations of this study, a preliminary experiment was carried out to choose a prompt-engineering-technique to test on the models.

In this experiment, GPT-3.5-turbo was run with six prompts based on different prompting-techniques (all found in Appendix **??**):

1. No Prompt Engineering
2. Basic
3. Chain of Thoughts
4. Chain of Verification
5. Tree of Thoughts
6. Graph of Thoughts

The statistical setup used for the experiments was identical for all dataset-creation-techniques (Section 3.1). Namely, 500 entries were generated for the training dataset. A total of 588 samples were used for the validation and testing set, with a split of 50%.

3.3.2 Data

3.3.3 Metrics

Evaluation of code synthesis has been becoming increasingly important due to the large amount of code generation models being introduced (). Human evaluation of this code is infeasible, and therefore a variety of metrics have been used for evaluation. However, as of yet there are few metrics specifically constructed for this task. Therefor, metrics that are generally used for natural language translation have been adopted. BLEU score (78) (Equation 4) is the most popular metric for evaluation of code synthesis.

$$\text{BLEU} = \text{BP} \times \exp\left(\sum_{n=1}^{N} \frac{1}{N} \log \text{precision}_n\right) \tag{4}$$

It is based on n-gram precision ($precision_n$) and uses brevity penalty (BR) for candidate translations shorter then the reference translation. Moreover, metrics like METEOR (25), CHRf (83), ROUGE-L (60) have also seen use. Two examples of metrics that are specifically developed to estimate the similarity of code are RUBY (105) and CodeBLEU (85), of which the latter more popular. CodeBLEU (Equation 5) is an extension of the BLEU score, which attaches weights to terms that are meaningful to the programming language ($BLEU_{weighted}$), keeps into account syntactic similarities and differences (AST_{match}) and checks for semantic differences through building a data flow out of the code (DFG_{match}). RUBY also considers the lexical, syntactical, and semantic representations of code, but does this through trying to build a graph representation of the code. If this is deemed impossible it falls back on an AST_{match} and if that fails, simple string similarity is checked.

$$CodeBLEU = \alpha \cdot BLEU + \beta \cdot BLEU_{weighted} + \gamma \cdot AST_{match} + \delta \cdot DFG_{match} \tag{5}$$

For ORCA input files .....

3.3.4 Hyperparameters

As for the finetuned models, a grid search was performed for every model configuration. Namely, because they are training on different datasets, and even tasks when prompt-engineering is used and thus the training landscape is supposedly different. The grid search involved the following options:

- **Epochs**: $\{1, 3, 5, 8, 10\}$
- **Batch Sizes**: $\{1, 2, 3, 8, 32, 64\}$
- η **multiplier**: $\{0.1, 0.2, 0.5, 0.75, 1, 2\}$
- **K**: $\{1, 5, 10, \textit{Limit}\}$

Unfortunately, the loss of GPT-3.5-turbo did not reflect our task well. During experimentation it was found that a low validation loss had no correlation with better validation results. Therefore, instead of using the validation loss as guidance as to what model performed best, we ran the models with the different configurations on the validation set ourselves and chose the one with the highest F1-score for further evaluation.

The chosen hyperparameters for the different models are shown in Table 1. The temperature of the models was always set to 1, and the top-1 prediction was always chosen.

Table 1: The used hyperparameters for the different models.

Model	**Epochs**	**Batch Size**	**η multiplier**	**K**
Base	✗	✗	✗	10
Base-COT	✗	✗	✗	10
Brute-force-500	5	1	2	5
Brute-force-500-COT	8	1	2	5
Manual-based-500	3	1	2	1
Manual-based-500-COT	8	1	2	1
Rule-based-500	3	1	2	1
Rule-based-500-COT	5	1	2	1

4 Results

Table 2: Comparison of GPT-3.5-turbo base model

COT	**RAG**	**Runnable (%)**	**F1**$_{total}$	**F1**$_{avg}$	**BLEU Score**
✗	✗	3.401	0.217	0.182	10.121
✗	✓	11.905	**0.242**	**0.194**	8.467
✓	✗	3.741	0.229	0.193	**10.288**
✓	✓	**14.626**	0.229	0.188	8.026

Table 3: Comparison of finetuned models

COT	**RAG**	**Finetuning Dataset**	**Runnable (%)**	**F1**$_{total}$	**F1**$_{avg}$	**BLEU Score**
✗	✗	Rule-based	11.225	0.333	0.237	8.962
✗	✓	Rule-based	10.884	0.282	0.206	7.311
✓	✗	Rule-based	19.048	0.361	0.254	9.418
✓	✓	Rule-based	15.646	0.298	0.205	7.504
✗	✗	Manual-based	10.544	0.390	0.261	12.293
✗	✓	Manual-based	14.626	0.311	0.228	7.887
✓	✗	**Manual-based**	21.769	**0.426**	**0.286**	**12.916**
✓	✓	Manual-based	15.986	0.330	0.223	8.300
✗	✗	Brute-force	15.306	0.382	0.267	10.234
✗	✓	Brute-force	25.122	0.221	0.177	10.963
✓	✗	Brute-force	17.006	0.384	0.264	17.114
✓	✓	Brute-force	**26.191**	0.282	0.205	7.312

5 Discussion

5.1 Conclusions

5.2 Future Research

Appendix

A Prompts

A.1 Prompt Improvement

```
**Goal:** Generate a natural language prompt for an ORCA input file
   based on user-provided descriptions.
**User Input:** List of keywords (separated by @), including basis
   set/functional types if applicable.
The keyword is placed first ant then followed by a '=' and its
   description.
Advanced settings for input blocks are within %...% blocks, where the
    description of the option keyword is placed first and the option
   itsself is placed within brackets.
This is then followed by the settings for that option. Lastly, a
   smiles for the molecule is provided after a '#'. \nStep 1: What
   molecule is used and what is its smiles?
Step 2: What keywords did the user provide, and what do they mean?
Step 3: Which of the keywords are the basis sets, the density
   functionals and which belong to other groups?
Step 4: Are there any advanced settings provided within %...% blocks
   (identify settings) and if so, what do they do? If there are none,
    you can write this down.
Step 5: Based on the identified information, compose a natural
   language prompt for the ORCA input file using your domain
   knowledge of chemistry and quantum physics by interpreting domain
   specific terms correctly.
Please preface and end the prompt you create by the string !PROMPT!.
An example format of your final output is:
!PROMPT! Perform a (type of calculation) with the (basis sets
   exaplanation) basis set, using the [molecule] molecule. Use the
   following settings: (settings). !PROMPT!
Please vary in the language you use in the prompt, make sure that it
   varies over different outputs and reflects how a user might ask
   you for an input file.
```

A.2 RAG

```
Note that the user will also provide you with contextual information
   out of ORCA manuals that could be relevant to the simulation that
   is desired. This contextual information is preceded by the
   following tag: #context
You can use this information in assisting you to form the input file
   the user desires, but be mindful to only use the context when it
   is actually helpful and relates to the users request.
Try to use this information to help you find out what keywords the
   user wants to use and how the input blocks of the input file
   should look.
Write down how and what parts of the contextual information you used
   to get to what part of the input file.
If you did not find anything useful, write this down as well.
```

A.3 Basic

```
Imagine you are a chemistry expert made to generate input files for
   the chemistry simulation language ORCA.
Given a prompt about the type of simulation the user wants, you
   generate an input file for ORCA,
however these input files don't contain the standard xyz coordinates.

The format of an ORCA input file is as follows:
!keyword keyword keyword
%option
setting_of_option value
setting_of_option2 value2
end
#smiles_of_molecule

In the line that starts with an ! you can define methods, density
   functionals and basis sets through ORCA keywords.
Then you can use input blocks to set certain settings.
This block starts with % and then the option and is followed by the
   settings within that input block. It ends with 'end'.
Lastly, you write down the smiles format of the molecule after a #.

Below are some example user prompts with corresponding input files to
    better illustrate the format at hand. Make sure to not overly
   rely on the keywords and settings in the input files when
   predicting a new input file.
```

```
Prompt 1:
Perform a quantum chemical calculation using the def2-svp basis set
   with new polarization functions and the def2/j basis set with
   Weigends universal Coulomb fitting, suitable for all def2 type
   basis sets. The calculation will be conducted with the Becke '88
   exchange functional and Perdew '86 correlation functional. Use O=[
   SH] as molecule.
Input file 1:
!bp86 def2-svp def2/j
%scf
maxiter 100
end
#O=[SH]

Prompt 2:
Perform a closed-shell SCF calculation with the def2-TZVPP basis set
   for property calculations and an automatically constructed
   auxiliary basis for fitting Coulomb, exchange, and correlation
   calculations. Use the RIJCOSX algorithm for efficient treatment of
   Coulomb and Exchange terms via RI and seminumerical integration,
   along with sloppy SCF convergence criteria. Optimize the geometry
   with GDIIS for normal optimization and loose convergence criteria.
   Avoid starting from an existing GBW file. Additionally, control
   the SCF procedure with forced convergence and relax the density
   for the MP2 calculation. Use B(hashtag)N as molecule.
Input file 2:
!def2-tzvppd rhf rijcosx sloppyscf autoaux gdiis-opt looseopt
   noautostart
%scf ConvForced true end
%mp2 Density relaxed end
%geom
Calc_Hess true
Recalc_Hess 1
end
#B(hashtag)N

Prompt 3:
Perform a calculation using the HartreeFock method and the Valence
   double-zeta basis set with "new" polarization functions for the O
   molecule.
Input file 3:
!hf def2-svp
```

```
#O

Prompt 4:
Perform a quantum mechanical calculation using the cc−pVTZ and cc−
   pVTZ/c basis sets for the S molecule. Additionally, include the
   advanced settings for the MP2 calculation, ensuring density
   relaxation and donatorbs. Implement the Local MP2 method for the
   calculation.
Input file 4:
!dlpno−mp2 cc−pvtz cc−pvtz/c
%mp2
density relaxed
donatorbs true
end
#S

Prompt 5:
Conduct a geometry optimization calculation on the False molecule
   using the popular B3LYP functional with a 20% HF exchange for the
   O molecule. For this, employ the valence triple−zeta basis set (
   def2−tzvp) with "new" polarization functions, similar to TZVPP for
    main group elements and TZVP for hydrogen. Additionally, control
   the frequency calculations with a temperature range of 290 K to
   300 K.
Input file 5:
!b3lyp def2−tzvp opt freq
%freq
    temp 290, 295, 300
end
#O
```

A.4 Chain of Thoughts

```
Imagine you are a chemistry expert made to generate input files for
   the chemistry simulation language ORCA.
Given a prompt about the type of simulation the user wants, you
   generate an input file for ORCA,
however these input files don't contain the standard xyz coordinates.
    Employ the Chain of Thoughts (CoT)
method to systematically navigate through the process of creating an
   accurate input file.

....
```

```
In making your input file, use step-by-step reasoning and write your
   reasoning down:
Step 1: Identify what ORCA keywords should be used, like for instance
    the basis set and a possible density functional.
Step 2: What are the input block options that should be used based on
    the user description and chosen keywords, if there are any.
Step 3: What are the ORCA input block settings needed in these input
   blocks to get to the desired settings?
Step 4: What is the SMILES of the molecule the user wants to use for
   these keywords and input blocks?
Step 5: What is the final ORCA input file?
Generate the ORCA input file by following these steps and ensure each
    step is explained.
```

A.5 Chain of Verification

```
Imagine you are a chemistry expert tasked with generating input files
    for the chemistry simulation language ORCA.
Given a prompt about the type of simulation the user wants, you
   generate an input file for ORCA,
however these input files don't contain the standard xyz coordinates.
To ensure the accuracy and correctness of the generated input file,
   you will use a Chain of Verification approach.
This method involves creating the input file and then verifying each
   component step-by-step to confirm its validity.

....

Given a prompt, start by creating an initial version of the input
   file based on the user's description.
Then verify it with these steps:
Step 1: Ensure the simulation type (e.g., geometry optimization,
   frequency calculation) matches the user's goal.
Step 2: Check that the selected methods, density functionals, and
   basis sets are appropriate for the simulation.
Step 3: Confirm that any advanced settings and input block options
   are correctly specified and relevant.
Step 4: Ensure the SMILES format of the molecule is correct and
   accurately represents the molecule.
Step 5: Conduct a comprehensive final review of the entire input file
    to ensure all components are coherent and correctly formatted.
```

```
Using this approach, generate the ORCA input file and verify each
   step to ensure the highest accuracy and alignment with the user's
   goals.
```

A.6 Graph of Thoughts

```
Imagine you are a chemistry expert tasked with generating input files
    for the chemistry simulation language ORCA.
Given a prompt about the type of simulation the user wants, you
   generate an input file for ORCA,
however these input files don't contain the standard xyz coordinates.
To ensure a thorough and comprehensive approach, you will use a Graph
    of Thoughts method.
This involves mapping out your reasoning in a non-linear,
   interconnected way,
exploring various possibilities and ensuring all aspects of the task
   are considered.

....

Use the following steps to generate the input file:
Step 1: Identify the core elements of the input file: simulation type
   , ORCA Keywords, Advanced Settings, Molecule Format
Step 2: Create connections between these elements, exploring how each
    decision impacts the others.
Step 3: For each element, consider various options and their
   implications.
Step 4: Combine the most suitable options into a coherent input file.
Generate the input file by following these steps and ensure each
   decision is justified and explained.
```

A.7 Tree of Thoughts

```
Imagine you are a chemistry expert tasked with generating input files
    for the chemistry simulation language ORCA.
Given a prompt about the type of simulation the user wants, you
   generate an input file for ORCA,
however these input files don't contain the standard xyz coordinates.
Your goal is to create a comprehensive and accurate input file based
   on the user's description of the desired simulation.
You will approach this task by exploring different branches of
   thought and consolidating the best ideas.
```

```
.....
Use a tree of thoughts process to generate the input file:
1. Identify the type of simulation (e.g., geometry optimization,
   frequency calculation, etc.).
Determine the appropriate ORCA keywords (e.g., methods, density
   functionals, basis sets).
Consider advanced settings and input block options.
2. Evaluate each initial thought for relevance and accuracy. Select
   the most appropriate keywords and settings.
3. Consolidate the best options into a coherent input file. Review
   and refine to ensure accuracy and completeness.
Generate the input file by following these steps and ensure each step
    is explained.
```

References

[1] Aggarwal, A., Chauhan, A., Kumar, D., Mittal, M., and Verma, S. (2020). Classification of fake news by fine-tuning deep bidirectional transformers based language model. page 163973.

[2] Ahmad, W., Simon, E., Chithrananda, S., Grand, G., and Ramsundar, B. (2022). Chemberta-2: Towards chemical foundation models.

[3] Allamanis, M., Tarlow, D., Gordon, A., and Wei, Y. (2015). Bimodal modelling of source code and natural language. In Bach, F. and Blei, D., editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 2123–2132, Lille, France. PMLR.

[4] Alon, U., Brody, S., Levy, O., and Yahav, E. (2018). code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400.*

[5] Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., and Sutton, C. (2021). Program synthesis with large language models.

[6] Bauer, A., Trapp, S., Stenger, M., Leppich, R., Kounev, S., Leznik, M., Chard, K., and Foster, I. (2024). Comprehensive exploration of synthetic data generation: A survey.

[7] Baumann, N., Diaz, J. S., Michael, J., Netz, L., Nqiri, H., Reimer, J., and Rumpe, B. (2024). Combining retrieval-augmented generation and few-shot learning for model synthesis of uncommon dsls. Modellierung 2024 Satellite Events.

[8] Bavishi, R., Lemieux, C., Sen, K., and Stoica, I. (2021). Gauss: program synthesis by reasoning over graphs. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–29.

[9] Besta, M., Blach, N., Kubicek, A., Gerstenberger, R., Podstawski, M., Gianinazzi, L., Gajda, J., Lehmann, T., Niewiadomski, H., Nyczyk, P., and Hoefler, T. (2024). Graph of thoughts: Solving elaborate problems with large language models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(16):17682–17690.

[10] Boiko, D. A., MacKnight, R., Kline, B., and Gomes, G. (2023). Autonomous chemical research with large language models. *Nature*, 624(7992):570–578.

[11] Bran, A. M., Cox, S., Schilter, O., Baldassari, C., White, A. D., and Schwaller, P. (2023). Chemcrow: Augmenting large-language models with chemistry tools.

[12] Brown, T. B. et al. (2020a). Language models are few-shot learners. *CoRR*, abs/2005.14165.

[13] 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., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020b). Language models are few-shot learners.

[14] Chaudhary, S. (2023). Code alpaca: An instruction-following llama model for code generation. `https://github.com/sahil280114/codealpaca`.

[15] Chen, B., Zhang, Z., Langrené, N., and Zhu, S. (2023a). Unleashing the potential of prompt engineering in large language models: a comprehensive review.

[16] Chen, J., Chen, L., Huang, H., and Zhou, T. (2023b). When do you need chain-of-thought prompting for chatgpt? *arXiv preprint arXiv:2304.03262*.

[17] Chen, M. et al. (2021). Evaluating large language models trained on code.

[18] Chiang, W.-L., Zheng, L., Sheng, Y., Angelopoulos, A. N., Li, T., Li, D., Zhang, H., Zhu, B., Jordan, M., Gonzalez, J. E., and Stoica, I. (2024). Chatbot arena: An open platform for evaluating llms by human preference.

[19] Chithrananda, S., Grand, G., and Ramsundar, B. (2020). Chemberta: large-scale self-supervised pretraining for molecular property prediction. *arXiv preprint arXiv:2010.09885*.

[20] Chokwitthaya, C., Zhu, Y., Mukhopadhyay, S., and Jafari, A. (2020). Applying the gaussian mixture model to generate large synthetic data from a small data set. In *Construction Research Congress 2020*, pages 1251–1260. American Society of Civil Engineers Reston, VA.

[21] Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., Garcia, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim, H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A. M.,

Pillai, T. S., Pellat, M., Lewkowycz, A., Moreira, E., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Diaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K., Eck, D., Dean, J., Petrov, S., and Fiedel, N. (2022). Palm: Scaling language modeling with pathways.

[22] Clement, C. B., Drain, D., Timcheck, J., Svyatkovskiy, A., and Sundaresan, N. (2020). Pymt5: multi-mode translation of natural language and python code with transformers. *CoRR*, abs/2010.03150.

[23] Dainese, N., Merler, M., Alakuijala, M., and Marttinen, P. (2024). Generating code world models with large language models guided by monte carlo tree search. *arXiv preprint arXiv:2405.15383*.

[24] DeepSeek-AI, Liu, A., Feng, B., Wang, B., Wang, B., Liu, B., Zhao, C., Dengr, C., Ruan, C., Dai, D., Guo, D., Yang, D., Chen, D., Ji, D., Li, E., Lin, F., Luo, F., Hao, G., Chen, G., Li, G., Zhang, H., Xu, H., Yang, H., Zhang, H., Ding, H., Xin, H., Gao, H., Li, H., Qu, H., Cai, J. L., Liang, J., Guo, J., Ni, J., Li, J., Chen, J., Yuan, J., Qiu, J., Song, J., Dong, K., Gao, K., Guan, K., Wang, L., Zhang, L., Xu, L., Xia, L., Zhao, L., Zhang, L., Li, M., Wang, M., Zhang, M., Zhang, M., Tang, M., Li, M., Tian, N., Huang, P., Wang, P., Zhang, P., Zhu, Q., Chen, Q., Du, Q., Chen, R. J., Jin, R. L., Ge, R., Pan, R., Xu, R., Chen, R., Li, S. S., Lu, S., Zhou, S., Chen, S., Wu, S., Ye, S., Ma, S., Wang, S., Zhou, S., Yu, S., Zhou, S., Zheng, S., Wang, T., Pei, T., Yuan, T., Sun, T., Xiao, W. L., Zeng, W., An, W., Liu, W., Liang, W., Gao, W., Zhang, W., Li, X. Q., Jin, X., Wang, X., Bi, X., Liu, X., Wang, X., Shen, X., Chen, X., Chen, X., Nie, X., Sun, X., Wang, X., Liu, X., Xie, X., Yu, X., Song, X., Zhou, X., Yang, X., Lu, X., Su, X., Wu, Y., Li, Y. K., Wei, Y. X., Zhu, Y. X., Xu, Y., Huang, Y., Li, Y., Zhao, Y., Sun, Y., Li, Y., Wang, Y., Zheng, Y., Zhang, Y., Xiong, Y., Zhao, Y., He, Y., Tang, Y., Piao, Y., Dong, Y., Tan, Y., Liu, Y., Wang, Y., Guo, Y., Zhu, Y., Wang, Y., Zou, Y., Zha, Y., Ma, Y., Yan, Y., You, Y., Liu, Y., Ren, Z. Z., Ren, Z., Sha, Z., Fu, Z., Huang, Z., Zhang, Z., Xie, Z., Hao, Z., Shao, Z., Wen, Z., Xu, Z., Zhang, Z., Li, Z., Wang, Z., Gu, Z., Li, Z., and Xie, Z. (2024). Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model.

[25] Denkowski, M. and Lavie, A. (2014). Meteor universal: Language specific translation evaluation for any target language. In *Proceedings of the ninth workshop on statistical machine translation*, pages 376–380.

[26] Desai, A., Gulwani, S., Hingorani, V., Jain, N., Karkare, A., Marron, M., and Roy, S. (2016). Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, pages 345–356.

[27] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding.

[28] Dhuliawala, S., Komeili, M., Xu, J., Raileanu, R., Li, X., Celikyilmaz, A., and Weston, J. (2023). Chain-of-verification reduces hallucination in large language models.

[29] Dodge, J., Ilharco, G., Schwartz, R., Farhadi, A., Hajishirzi, H., and Smith, N. (2020). Fine-tuning pretrained language models: Weight initializations, data orders, and early stopping.

[30] Dong, G., Yuan, H., Lu, K., Li, C., Xue, M., Liu, D., Wang, W., Yuan, Z., Zhou, C., and Zhou, J. (2024). How abilities in large language models are affected by supervised fine-tuning data composition.

[31] Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P.-E., Lomeli, M., Hosseini, L., and Jégou, H. (2024). The faiss library.

[32] Eigenschink, P., Reutterer, T., Vamosi, S., Vamosi, R., Sun, C., and Kalcher, K. (2023). Deep generative models for synthetic sequential data: A survey. *IEEE Access*.

[33] Enis, M. and Hopkins, M. (2024). From llm to nmt: Advancing low-resource machine translation with claude.

[34] Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., and Zhang, J. M. (2023). Large language models for software engineering: Survey and open problems.

[35] Feng, S. Y., Gangal, V., Wei, J., Chandar, S., Vosoughi, S., Mitamura, T., and Hovy, E. (2021). A survey of data augmentation approaches for nlp. *arXiv preprint arXiv:2105.03075*.

[36] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. (2020). Codebert: A pre-trained model for programming and natural languages. *CoRR*, abs/2002.08155.

[37] Frisch, M. J., Trucks, G. W., Schlegel, H. B., Scuseria, G. E., Robb, M. A., Cheeseman, J. R., Scalmani, G., Barone, V., Petersson, G. A., Nakatsuji, H., Li, X., Caricato, M., Marenich, A. V., Bloino, J., Janesko, B. G., Gomperts, R., Mennucci, B., Hratchian, H. P., Ortiz, J. V., Izmaylov, A. F., Sonnenberg, J. L., Williams-Young, D., Ding, F., Lipparini, F., Egidi, F., Goings, J., Peng, B., Petrone, A., Henderson, T., Ranasinghe, D., Zakrzewski, V. G., Gao, J., Rega, N., Zheng, G., Liang, W., Hada, M., Ehara, M., Toyota, K., Fukuda, R., Hasegawa, J., Ishida, M., Nakajima, T., Honda, Y., Kitao, O., Nakai, H., Vreven, T., Throssell, K., Montgomery, Jr., J. A., Peralta, J. E., Ogliaro, F., Bearpark, M. J., Heyd, J. J., Brothers, E. N., Kudin, K. N., Staroverov, V. N., Keith, T. A., Kobayashi, R., Normand, J., Raghavachari, K., Rendell, A. P., Burant, J. C., Iyengar, S. S., Tomasi, J., Cossi, M., Millam, J. M., Klene, M., Adamo, C., Cammi, R., Ochterski, J. W., Martin, R. L., Morokuma, K., Farkas, O., Foresman, J. B., and Fox, D. J. (2016). Gaussian˜16 Revision C.01. Gaussian Inc. Wallingford CT.

[38] Gao, X., Kim, T., Wong, M. D., Raghunathan, D., Varma, A. K., Kannan, P. G., Sivaraman, A., Narayana, S., and Gupta, A. (2020). Switch code generation using program synthesis. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 44–61.

[39] Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., and Wang, H. (2023). Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*.

[40] Gehman, S., Gururangan, S., Sap, M., Choi, Y., and Smith, N. A. (2020). Realtoxicityprompts: Evaluating neural toxic degeneration in language models. In *Findings*.

[41] Gekhman, Z., Yona, G., Aharoni, R., Eyal, M., Feder, A., Reichart, R., and Herzig, J. (2024). Does fine-tuning llms on new knowledge encourage hallucinations?

[42] Gulwani, S., Polozov, O., Singh, R., et al. (2017). Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119.

[43] Hendler, J. A., Tate, A., and Drummond, M. (1990). Ai planning: Systems and techniques. *AI magazine*, 11(2):61–61.

[44] Hocky, G. M. and White, A. D. (2022). Natural language processing models that automate programming will transform chemistry research and teaching. *Digital discovery*, 1(2):79–83.

[45] Houkjær, K., Torp, K., and Wind, R. (2006). Simple and realistic data generation. In *Proceedings of the 32nd international conference on Very large data bases*, pages 1243–1246.

[46] Huang, J., Gu, S. S., Hou, L., Wu, Y., Wang, X., Yu, H., and Han, J. (2022). Large language models can self-improve.

[47] Jiang, J., Ren, L., Xiong, Y., and Zhang, L. (2019). Inferring program transformations from singular examples via big code. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 255–266. IEEE.

[48] Kaddour, J., Harris, J., Mozes, M., Bradley, H., Raileanu, R., and McHardy, R. (2023). Challenges and applications of large language models.

[49] Karia, V., Mukherjee, A., Doshi, Z., Pendse, V., and Chang, V. (2019). Code2cap: Automated code captioning.

[50] Karpukhin, V., Oğuz, B., Min, S., Lewis, P., Wu, L., Edunov, S., Chen, D., and Yih, W.-t. (2020). Dense passage retrieval for open-domain question answering. *arXiv preprint arXiv:2004.04906*.

[51] Kaufmann, T., Weng, P., Bengs, V., and Hüllermeier, E. (2024). A survey of reinforcement learning from human feedback.

[52] Kaur, J. N., Bhatia, S., Aggarwal, M., Bansal, R., and Krishnamurthy, B. (2022). Lm-core: Language models with contextually relevant external knowledge. *arXiv preprint arXiv:2208.06458*.

[53] Kim, G., Kim, S., Jeon, B., Park, J., and Kang, J. (2023). Tree of clarifications: Answering ambiguous questions with retrieval-augmented large language models. *arXiv preprint arXiv:2310.14696*.

[54] Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., and Iwasawa, Y. (2022). Large language models are zero-shot reasoners.

[Koul] Koul, N. *Prompt Engineering for Large Language Models*. Nimrita Koul.

[56] Landrum, G. (2013). Rdkit documentation. *Release*, 1(1-79):4.

[57] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., et al. (2020). Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474.

[58] Li, C., Liang, J., Zeng, A., Chen, X., Hausman, K., Sadigh, D., Levine, S., Fei-Fei, L., Xia, F., and Ichter, B. (2023a). Chain of code: Reasoning with a language model-augmented code emulator.

[59] Li, J., Li, G., Li, Y., and Jin, Z. (2023b). Structured chain-of-thought prompting for code generation.

[60] Lin, C.-Y. (2004). Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

[61] Liu, F., Liu, Y., Shi, L., Huang, H., Wang, R., Yang, Z., and Zhang, L. (2024). Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971*.

[62] Liu, S., Nie, W., Wang, C., Lu, J., Qiao, Z., Liu, L., Tang, J., Xiao, C., and Anandkumar, A. (2022). Multi-modal molecule structure-text model for text-based retrieval and editing.

[63] Liu, X., Ji, K., Fu, Y., Du, Z., Yang, Z., and Tang, J. (2021). P-tuning v2: Prompt tuning can be comparable to fine-tuning universally across scales and tasks. *CoRR*, abs/2110.07602.

[64] Lu, S., Duan, N., Han, H., Guo, D., won Hwang, S., and Svyatkovskiy, A. (2022). Reacc: A retrieval-augmented code completion framework.

[65] Lu, Y., Bartolo, M., Moore, A., Riedel, S., and Stenetorp, P. (2021). Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. *CoRR*, abs/2104.08786.

[66] Mehrafarin, H., Rajaee, S., and Pilehvar, M. T. (2022). On the importance of data size in probing fine-tuned models.

[67] Min, B., Ross, H., Sulem, E., Veyseh, A. P. B., Nguyen, T. H., Sainz, O., Agirre, E., Heinz, I., and Roth, D. (2021). Recent advances in natural language processing via large pre-trained language models: A survey.

[68] Mumuni, A. and Mumuni, F. (2022). Data augmentation: A comprehensive survey of modern approaches. *Array*, 16:100258.

[69] Neese, F., Wennmohs, F., Becker, U., and Riplinger, C. (2020). The orca quantum chemistry program package. *The Journal of chemical physics*, 152(22).

[70] Ngassom, S. K., Dakhel, A. M., Tambon, F., and Khomh, F. (2024). Chain of targeted verification questions to improve the reliability of code generated by llms. *arXiv preprint arXiv:2405.13932*.

[71] Ni, A., Iyer, S., Radev, D., Stoyanov, V., tau Yih, W., Wang, S. I., and Lin, X. V. (2023). Lever: Learning to verify language-to-code generation with execution.

[72] Noonan, R. E. (1985). An algorithm for generating abstract syntax trees. *Computer Languages*, 10(3):225–236.

[73] OBrien, D., Biswas, S., Imtiaz, S. M., Abdalkareem, R., Shihab, E., and Rajan, H. (2024). Are prompt engineering and todo comments friends or foes? an evaluation on github copilot. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

[74] Ogueji, K., Zhu, Y., and Lin, J. (2021). Small data? no problem! exploring the viability of pretrained multilingual language models for low-resourced languages. In Ataman, D., Birch, A., Conneau, A., Firat, O., Ruder, S., and Sahin, G. G., editors, *Proceedings of the 1st Workshop on Multilingual Representation Learning*, pages 116–126, Punta Cana, Dominican Republic. Association for Computational Linguistics.

[75] OpenAI (2023). Gpt-4 technical report.

[76] Ouyang, L. et al. (2022). Training language models to follow instructions with human feedback.

[77] Pan, S., Luo, L., Wang, Y., Chen, C., Wang, J., and Wu, X. (2024). Unifying large language models and knowledge graphs: A roadmap. *IEEE Transactions on Knowledge and Data Engineering*.

[78] Papineni, K., Roukos, S., Ward, T., and Zhu, W. J. (2002). Bleu: a method for automatic evaluation of machine translation.

[79] Pargas, R. P., Harrold, M. J., and Peck, R. R. (1999). Test-data generation using genetic algorithms. *Software testing, verification and reliability*, 9(4):263–282.

[80] Parisotto, E., rahman Mohamed, A., Singh, R., Li, L., Zhou, D., and Kohli, P. (2016). Neuro-symbolic program synthesis.

[81] Parvez, M. R., Ahmad, W. U., Chakraborty, S., Ray, B., and Chang, K.-W. (2021). Retrieval augmented code generation and summarization.

[82] Peng, B., Galley, M., He, P., Cheng, H., Xie, Y., Hu, Y., Huang, Q., Liden, L., Yu, Z., Chen, W., and Gao, J. (2023). Check your facts and try again: Improving large language models with external knowledge and automated feedback.

[83] Popović, M. (2015). chrf: character n-gram f-score for automatic mt evaluation. In *Proceedings of the tenth workshop on statistical machine translation*, pages 392–395.

[84] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. (2019). Exploring the limits of transfer learning with a unified text-to-text transformer.

[85] Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., and Ma, S. (2020). Codebleu: a method for automatic evaluation of code synthesis. *CoRR*, abs/2009.10297.

[86] Ridnik, T., Kredo, D., and Friedman, I. (2024). Code generation with alphacodium: From prompt engineering to flow engineering.

[87] Rolim, R., Soares, G., D'Antoni, L., Polozov, O., Gulwani, S., Gheyi, R., Suzuki, R., and Hartmann, B. (2017). Learning syntactic program transformations from examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 404–415. IEEE.

[88] Sahoo, P., Singh, A. K., Saha, S., Jain, V., Mondal, S., and Chadha, A. (2024). A systematic survey of prompt engineering in large language models: Techniques and applications.

[89] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

[90] Shao, Z., Gong, Y., Shen, Y., Huang, M., Duan, N., and Chen, W. (2023). Synthetic prompting: Generating chain-of-thought demonstrations for large language models. In *International Conference on Machine Learning*, pages 30706–30775. PMLR.

[91] Shibata, Y., Kida, T., Fukamachi, S., Takeda, M., Shinohara, A., Shinohara, T., and Arikawa, S. (1999). Byte pair encoding: A text compression scheme that accelerates pattern matching.

[92] Shin, R., Kant, N., Gupta, K., Bender, C., Trabucco, B., Singh, R., and Song, D. (2019). Synthetic datasets for neural program synthesis.

[93] Shin, T., Razeghi, Y., IV, R. L. L., Wallace, E., and Singh, S. (2020). Autoprompt: Eliciting knowledge from language models with automatically generated prompts. *CoRR*, abs/2010.15980.

[94] Shuster, K., Poff, S., Chen, M., Kiela, D., and Weston, J. (2021a). Retrieval augmentation reduces hallucination in conversation. *arXiv preprint arXiv:2104.07567*.

[95] Shuster, K., Poff, S., Chen, M., Kiela, D., and Weston, J. (2021b). Retrieval augmentation reduces hallucination in conversation. *arXiv preprint arXiv:2104.07567*.

[96] Skreta, M., Yoshikawa, N., Arellano-Rubach, S., Ji, Z., Kristensen, L. B., Darvish, K., Aspuru-Guzik, A., Shkurti, F., and Garg, A. (2023). Errors are useful prompts: Instruction guided task programming with verifier-assisted iterative prompting.

[97] Slivkins, A. et al. (2019). Introduction to multi-armed bandits. *Foundations and Trends® in Machine Learning*, 12(1-2):1–286.

[98] Sun, Q., Berkelbach, T. C., Blunt, N. S., Booth, G. H., Guo, S., Li, Z., Liu, J., McClain, J. D., Sayfutyarova, E. R., Sharma, S., et al. (2018). Pyscf: the python-based simulations of chemistry framework. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 8(1):e1340.

[99] Tan, H., Luo, Q., Jiang, L., Zhan, Z., Li, J., Zhang, H., and Zhang, Y. (2024). Prompt-based code completion via multi-retrieval augmented generation.

[100] Team, G., Reid, M., Savinov, N., Teplyashin, D., Dmitry, Lepikhin, Lillicrap, T., baptiste Alayrac, J., Soricut, R., Lazaridou, A., Firat, O., Schrittwieser, J., Antonoglou, I., Anil, R., Borgeaud, S., Dai, A., Millican, K., Dyer, E., Glaese, M., Sottiaux, T., Lee, B., Viola, F., Reynolds, M., Xu, Y., Molloy, J., Chen, J., Isard, M., Barham, P., Hennigan, T., McIlroy, R., Johnson, M., Schalkwyk, J., Collins, E., Rutherford, E., Moreira, E., Ayoub, K., Goel, M., Meyer, C., Thornton, G., Yang, Z., Michalewski, H., Abbas, Z., Schucher, N., Anand, A., Ives, R., Keeling, J., Lenc, K., Haykal, S., Shakeri, S., Shyam, P., Chowdhery, A., Ring, R., Spencer, S., Sezener, E., Vilnis, L., Chang, O., Morioka, N., Tucker, G., Zheng, C., Woodman, O., Attaluri, N., Kocisky, T., Eltyshev, E., Chen, X., Chung, T.,

Selo, V., Brahma, S., Georgiev, P., Slone, A., Zhu, Z., Lottes, J., Qiao, S., Caine, B., Riedel, S., Tomala, A., Chadwick, M., Love, J., Choy, P., Mittal, S., Houlsby, N., Tang, Y., Lamm, M., Bai, L., Zhang, Q., He, L., Cheng, Y., Humphreys, P., Li, Y., Brin, S., Cassirer, A., Miao, Y., Zilka, L., Tobin, T., Xu, K., Proleev, L., Sohn, D., Magni, A., Hendricks, L. A., Gao, I., Ontanon, S., Bunyan, O., Byrd, N., Sharma, A., Zhang, B., Pinto, M., Sinha, R., Mehta, H., Jia, D., Caelles, S., Webson, A., Morris, A., Roelofs, B., Ding, Y., Strudel, R., Xiong, X., Ritter, M., Dehghani, M., Chaabouni, R., Karmarkar, A., Lai, G., Mentzer, F., Xu, B., Li, Y., Zhang, Y., Paine, T. L., Goldin, A., Neyshabur, B., Baumli, K., Levskaya, A., Laskin, M., Jia, W., Rae, J. W., Xiao, K., He, A., Giordano, S., Yagati, L., Lespiau, J.-B., Natsev, P., Ganapathy, S., Liu, F., Martins, D., Chen, N., Xu, Y., Barnes, M., May, R., Vezer, A., Oh, J., Franko, K., Bridgers, S., Zhao, R., Wu, B., Mustafa, B., Sechrist, S., Parisotto, E., Pillai, T. S., Larkin, C., Gu, C., Sorokin, C., Krikun, M., Guseynov, A., Landon, J., Datta, R., Pritzel, A., Thacker, P., Yang, F., Hui, K., Hauth, A., Yeh, C.-K., Barker, D., Mao-Jones, J., Austin, S., Sheahan, H., Schuh, P., Svensson, J., Jain, R., Ramasesh, V., Briukhov, A., Chung, D.-W., von Glehn, T., Butterfield, C., Jhakra, P., Wiethoff, M., Frye, J., Grimstad, J., Changpinyo, B., Lan, C. L., Bortsova, A., Wu, Y., Voigtlaender, P., Sainath, T., Gu, S., Smith, C., Hawkins, W., Cao, K., Besley, J., Srinivasan, S., Omernick, M., Gaffney, C., Surita, G., Burnell, R., Damoc, B., Ahn, J., Brock, A., Pajarskas, M., Petrushkina, A., Noury, S., Blanco, L., Swersky, K., Ahuja, A., Avrahami, T., Misra, V., de Liedekerke, R., Iinuma, M., Polozov, A., York, S., van den Driessche, G., Michel, P., Chiu, J., Blevins, R., Gleicher, Z., Recasens, A., Rrustemi, A., Gribovskaya, E., Roy, A., Gworek, W., Arnold, S. M. R., Lee, L., Lee-Thorp, J., Maggioni, M., Piqueras, E., Badola, K., Vikram, S., Gonzalez, L., Baddepudi, A., Senter, E., Devlin, J., Qin, J., Azzam, M., Trebacz, M., Polacek, M., Krishnakumar, K., yiin Chang, S., Tung, M., Penchev, I., Joshi, R., Olszewska, K., Muir, C., Wirth, M., Hartman, A. J., Newlan, J., Kashem, S., Bolina, V., Dabir, E., van Amersfoort, J., Ahmed, Z., Cobon-Kerr, J., Kamath, A., Hrafnkelsson, A. M., Hou, L., Mackinnon, I., Frechette, A., Noland, E., Si, X., Taropa, E., Li, D., Crone, P., Gulati, A., Cevey, S., Adler, J., Ma, A., Silver, D., Tokumine, S., Powell, R., Lee, S., Vodrahalli, K., Hassan, S., Mincu, D., Yang, A., Levine, N., Brennan, J., Wang, M., Hodkinson, S., Zhao, J., Lipschultz, J., Pope, A., Chang, M. B., Li, C., Shafey, L. E., Paganini, M., Douglas, S., Bohnet, B., Pardo, F., Odoom, S., Rosca, M., dos Santos, C. N., Soparkar, K., Guez, A., Hudson, T., Hansen, S., Asawaroengchai, C., Addanki, R., Yu, T., Stokowiec, W., Khan, M., Gilmer, J., Lee, J., Bostock, C. G., Rong, K., Caton, J., Pejman, P., Pavetic, F., Brown, G., Sharma, V., Lučić, M., Samuel, R., Djolonga, J., Mandhane, A., Sjösund, L. L., Buchatskaya, E., White, E., Clay, N., Jiang, J., Lim, H., Hemsley, R., Cankara, Z., Labanowski, J., Cao, N. D., Steiner, D., Hashemi, S. H., Austin, J., Gergely, A., Blyth, T., Stanton, J., Shivakumar, K., Siddhant, A., Andreassen, A., Araya, C., Sethi, N., Shivanna, R., Hand, S., Bapna, A., Khodaei, A., Miech, A., Tanzer, G., Swing, A., Thakoor, S., Aroyo, L., Pan, Z., Nado, Z., Sygnowski, J., Winkler, S., Yu, D., Saleh, M., Maggiore, L., Bansal, Y., Garcia, X., Kazemi, M., Patil, P., Dasgupta, I., Barr, I., Giang, M., Kagohara, T., Danihelka, I., Marathe, A., Feinberg, V., Elhawaty, M., Ghelani, N., Horgan, D., Miller, H., Walker, L., Tanburn, R., Tariq, M., Shrivastava, D., Xia, F., Wang, Q., Chiu, C.-C., Ashwood, Z., Baatarsukh, K., Samangooei, S., Kaufman, R. L., Alcober, F., Stjerngren, A., Komarek, P., Tsihlas, K., Boral, A., Comanescu, R., Chen, J., Liu, R., Welty, C., Bloxwich, D., Chen, C., Sun, Y., Feng, F., Mauger, M., Dotiwalla, X., Hellendoorn, V., Sharman, M., Zheng, I., Haridasan, K., Barth-Maron, G., Swanson, C., Rogozińska, D., Andreev, A., Rubenstein, P. K., Sang, R., Hurt, D., Elsayed, G., Wang, R., Lacey, D., Ilić, A., Zhao, Y., Iwanicki, A., Lince, A., Chen, A., Lyu, C., Lebsack, C., Griffith, J., Gaba, M., Sandhu, P., Chen, P., Koop, A., Rajwar, R., Yeganeh, S. H., Chang, S., Zhu, R., Radpour, S., Davoodi, E., Lei, V. I., Xu, Y., Toyama, D., Segal, C., Wicke, M., Lin, H., Bulanova, A., Badia, A. P., Rakićević, N., Sprechmann, P., Filos, A., Hou, S., Campos, V., Kassner, N., Sachan, D., Fortunato, M., Iwuanyanwu, C., Nikolaev, V., Lakshminarayanan, B., Jazayeri, S., Varadarajan, M., Tekur, C., Fritz, D., Khalman, M., Reitter, D., Dasgupta, K., Sarcar, S., Ornduff, T., Snaider, J., Huot, F., Jia, J., Kemp, R., Trdin, N., Vijayakumar, A., Kim, L., Angermueller, C., Lao, L., Liu, T., Zhang, H., Engel, D., Greene, S., White, A., Austin, J., Taylor, L., Ashraf, S., Liu, D., Georgaki, M., Cai, I., Kulizhskaya, Y., Goenka, S., Saeta, B., Xu, Y., Frank, C., de Cesare, D., Robenek, B., Richardson, H., Alnahlawi, M., Yew, C., Ponnapalli, P., Tagliasacchi, M., Korchemniy, A., Kim, Y., Li, D., Rosgen, B., Levin, K., Wiesner, J., Banzal, P., Srinivasan, P., Yu, H., Çağlar Ünlü, Reid, D., Tung, Z., Finchelstein, D., Kumar, R., Elisseeff, A., Huang, J., Zhang, M., Aguilar, R., Giménez, M., Xia, J., Dousse, O., Gierke, W., Yates, D., Jalan, K., Li, L., Latorre-Chimoto, E., Nguyen, D. D., Durden, K., Kallakuri, P., Liu, Y., Johnson, M., Tsai, T., Talbert, A., Liu, J., Neitz, A., Elkind, C., Selvi, M., Jasarevic, M., Soares, L. B., Cui, A., Wang, P., Wang, A. W., Ye, X., Kallarackal, K., Loher, L., Lam, H., Broder, J., Holtmann-Rice, D., Martin, N., Ramadhana, B., Shukla, M., Basu, S., Mohan, A., Fernando, N., Fiedel, N., Paterson, K., Li, H., Garg, A., Park, J., Choi, D., Wu, D., Singh, S., Zhang, Z., Globerson, A., Yu, L., Carpenter, J., de Chaumont Quitry, F., Radebaugh, C., Lin, C.-C., Tudor, A., Shroff, P., Garmon, D., Du, D., Vats, N., Lu, H., Iqbal, S., Yakubovich, A., Tripuraneni, N., Manyika, J., Qureshi, H., Hua, N., Ngani, C., Raad, M. A., Forbes, H., Stanway, J., Sundararajan, M., Ungureanu, V., Bishop, C., Li, Y., Venkatraman, B., Li, B., Thornton, C., Scellato, S., Gupta, N., Wang, Y., Tenney, I., Wu, X., Shenoy, A., Carvajal, G., Wright, D. G., Bariach, B., Xiao, Z., Hawkins, P., Dalmia, S., Farabet, C., Valenzuela, P., Yuan, Q., Agarwal, A., Chen, M., Kim, W., Hulse, B., Dukkipati, N., Paszke, A., Bolt, A., Choo, K., Beattie, J., Prendki, J., Vashisht, H., Santamaria-Fernandez, R., Cobo, L. C., Wilkiewicz, J., Madras, D., Elqursh, A., Uy, G., Ramirez, K., Harvey, M., Liechty, T., Zen, H., Seibert, J., Hu, C. H., Khorlin, A., Le, M., Aharoni, A., Li, M., Wang, L., Kumar, S., Casagrande, N., Hoover, J., Badawy, D. E., Soergel, D., Vnukov, D., Miecnikowski, M., Simsa, J., Kumar, P., Sellam, T., Vlasic, D., Daruki, S., Shabat, N., Zhang, J., Su, G., Zhang, J., Liu, J., Sun, Y., Palmer, E., Ghaffarkhah, A., Xiong, X., Cotruta, V., Fink, M., Dixon, L., Sreevatsa, A., Goedeckemeyer, A., Dimitriev, A., Jafari, M., Crocker, R., FitzGerald, N., Kumar, A., Ghemawat, S., Philips, I., Liu, F., Liang, Y., Sterneck, R., Repina, A., Wu, M., Knight, L., Georgiev, M., Lee, H., Askham, H., Chakladar, A., Louis, A., Crous, C., Cate, H., Petrova, D., Quinn, M., Owusu-Afriyie, D., Singhal, A., Wei, N., Kim, S., Vincent, D., Nasr, M., Choquette-Choo, C. A., Tojo, R., Lu, S., de Las Casas, D., Cheng, Y., Bolukbasi, T., Lee, K., Fatehi, S., Ananthanarayanan, R., Patel, M., Kaed, C., Li, J., Belle, S. R., Chen, Z., Konzelmann, J., Põder, S., Garg, R., Koverkathu, V., Brown, A., Dyer, C., Liu, R., Nova, A., Xu, J., Walton, A., Parrish, A., Epstein, M., McCarthy, S., Petrov, S., Hassabis, D., Kavukcuoglu, K., Dean, J., and Vinyals, O. (2024). Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context.

[101] Tian, Y., Yan, W., Yang, Q., Chen, Q., Wang, W., Luo, Z., and Ma, L. (2024). Codehalu: Code hallucinations in llms driven by execution-based verification.

[102] Tommasi, T., Patricia, N., Caputo, B., and Tuytelaars, T. (2017). A deeper look at dataset bias. *Domain adaptation in computer vision applications*, pages 37–55.

[103] Torralba, A. and Efros, A. A. (2011). Unbiased look at dataset bias. In *CVPR 2011*, pages 1521–1528. IEEE.

[104] 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., and Lample, G. (2023). Llama: Open and efficient foundation language models.

[105] Tran, N. M., Tran, H., Nguyen, S., Nguyen, H., and Nguyen, T. N. (2019). Does BLEU score work for code migration? *CoRR*, abs/1906.04903.

[106] Valmeekam, K., Marquez, M., Sreedharan, S., and Kambhampati, S. (2023). On the planning abilities of large language models-a critical investigation. *Advances in Neural Information Processing Systems*, 36:75993–76005.

[107] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention Is All You Need.

[108] Verhoef, P. C., Broekhuizen, T., Bart, Y., Bhattacharya, A., Dong, J. Q., Fabian, N., and Haenlein, M. (2021). Digital transformation: A multidisciplinary reflection and research agenda. *Journal of business research*, 122:889–901.

[109] Vulić, I., Ponti, E. M., Korhonen, A., and Glavaš, G. (2021). LexFit: Lexical fine-tuning of pretrained language models. In Zong, C., Xia, F., Li, W., and Navigli, R., editors, *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)*, pages 5269–5283, Online. Association for Computational Linguistics.

[110] Wang, C., Yang, Y., Gao, C., Peng, Y., Zhang, H., and Lyu, M. R. (2022). No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering*, pages 382–394.

[111] Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., Chen, Z., Tang, J., Chen, X., Lin, Y., et al. (2024). A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):1–26.

[112] Wang, S., Guo, Y., Wang, Y., Sun, H., and Huang, J. (2019). Smiles-bert: large scale unsupervised pre-training for molecular property prediction. In *Proceedings of the 10th ACM international conference on bioinformatics, computational biology and health informatics*, pages 429–436.

[113] Wang, W., Wang, Y., Joty, S., and Hoi, S. C. (2023). Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 146–158, New York, NY, USA. Association for Computing Machinery.

[114] Wang, X., Dillig, I., and Singh, R. (2017). Program synthesis using abstraction refinement. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–30.

[115] Wang, Y., Wang, W., Joty, S., and Hoi, S. C. (2021). Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

[116] Wei, J., Wang, X., Schuurmans, D., Bosma, M., ichter, b., Xia, F., Chi, E., Le, Q. V., and Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. In Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., and Oh, A., editors, *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837. Curran Associates, Inc.

[117] Weininger, D. (1988). Smiles, a chemical language and information system. 1. introduction to methodology and encoding rules. *Journal of chemical information and computer sciences*, 28(1):31–36.

[118] Weiss, K., Khoshgoftaar, T. M., and Wang, D. (2016). A survey of transfer learning. *Journal of Big data*, 3:1–40.

[119] White, A. D., Hocky, G. M., Gandhi, H. A., Ansari, M., Cox, S., Wellawatte, G. P., Sasmal, S., Yang, Z., Liu, K., Singh, Y., and Peña Ccoa, W. J. (2023). Assessment of chemistry knowledge in large language models that generate code. *Digital Discovery*, 2:368–376.

[120] Wilkins, D. E. and Myers, K. L. (1995). A common knowledge representation for plan generation and reactive execution. *Journal of Logic and Computation*, 5(6):731–761.

[121] Xu, F. F., Alon, U., Neubig, G., and Hellendoorn, V. J. (2022). A systematic evaluation of large language models of code.

[122] Xu, R., Luo, F., Zhang, Z., Tan, C., Chang, B., Huang, S., and Huang, F. (2021). Raise a child in large language model: Towards effective and generalizable fine-tuning.

[123] Yao, J.-Y., Ning, K.-P., Liu, Z.-H., Ning, M.-N., and Yuan, L. (2023a). Llm lies: Hallucinations are not bugs, but features as adversarial examples. *arXiv preprint arXiv:2310.01469*.

[124] Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., and Narasimhan, K. (2023b). Tree of thoughts: Deliberate problem solving with large language models.

[125] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. (2023c). React: Synergizing reasoning and acting in language models.

[126] Ye, J., Chen, X., Xu, N., Zu, C., Shao, Z., Liu, S., Cui, Y., Zhou, Z., Gong, C., Shen, Y., Zhou, J., Chen, S., Gui, T., Zhang, Q., and Huang, X. (2023). A comprehensive capability analysis of gpt-3 and gpt-3.5 series models.

[127] Yin, P. and Neubig, G. (2017). A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*.

[128] Yu, S., Wang, T., and Wang, J. (2022). Data augmentation by program transformation. *Journal of Systems and Software*, 190:111304.

[129] Yu, Z., He, L., Wu, Z., Dai, X., and Chen, J. (2023). Towards better chain-of-thought prompting strategies: A survey. *arXiv preprint arXiv:2310.04959*.

[130] Zhang, S., Dong, L., Li, X., Zhang, S., Sun, X., Wang, S., Li, J., Hu, R., Zhang, T., Wu, F., and Wang, G. (2024). Instruction tuning for large language models: A survey.

[131] Zheng, X., Ji, R., Chen, Y., Wang, Q., Zhang, B., Chen, J., Ye, Q., Huang, F., and Tian, Y. (2021). Migo-nas: Towards fast and generalizable neural architecture search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(9):2936–2952.

[132] Zhou, S., Alon, U., Xu, F. F., Wang, Z., Jiang, Z., and Neubig, G. (2022). Docprompting: Generating code by retrieving the docs. *arXiv preprint arXiv:2207.05987*.