Supplementary Materials: Developing Large Language Models for Quantum Chemistry Simulation Input Generation

Pieter Floris Jacobs[1]
Robert Pollice[1,*]

[1] Stratingh Institute for Chemistry, University of Groningen, Groningen, The Netherlands
* Corresponding author: r.pollice@rug.nl

1 Additional Related Work

1.1 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 molecular design. More specifically, they are used for 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.* [1] 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`[2], `ChemBERTa`[3], and `ChemBERTa-2`[4] are instances of LLMs that were all built to map the SMILES (Simplified Molecular Input Line Entry System) notation[5], 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 the previously mentioned one conducted by Hocky and White[6]. In their research, they utilized the `Codex` model[7] to generate input files for the Gaussian software package. 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 the `rdkit` library to get proper coordinates for the molecules. They found that the generated code rarely had syntax mistakes and showed that the model had some basic chemistry knowledge. However, it 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 general purpose LLM's 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. White *et al.*[8] assessed the general chemistry knowledge of various LLMs like `Codex` and both 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 manually 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.

Additionally, they discovered valuable prompt engineering strategies for future investigations. For instance, 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 more professional code.

Boiko *et al.*[9] 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, Documentation Searcher, Lab Automation, Code Execution. 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. M. Bran *et al.*[10] also proposed an even more sophisticated LLM chemistry agent called `ChemCrow` that is able to make use of 18 tools. These tools range from general utilities, such as web search, to specialized ones, including molecular tools that can convert molecule names to their SMILES representations and safety tools that assess whether a molecule is explosive or not. `ChemCrow` uses a ReAct[11] based workflow to reason about what tools to use given a user prompt. In short, this entails using a chain-of-thought reasoning[12] process with basic action plan generation[13–15].

1.2 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[16], making use of generative models[17] and making use of statistical models like Gaussian Mixture Modelling[18].

Direct generation of data for code synthesis has long been receiving academic attention, with early heuristics like genetic algorithms[19] and graph models[20] seeing some success. More modern approaches tend to make use of generative models to model the underlying data distribution of a programming language and sample from it to generate new data. Yin and Neubig[21], 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.* [22] 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.* [23] 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.* [24] 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 substantial generalisation performance across different programming languages. A very similar work by Karia *et al.* [25] changed the `code2sec` architecture slightly by incorporating a Transformer into it. Allamanis *et al.* [26] 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 two most recently mentioned 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.* [27].

1.3 Prompt Engineering

Prompt engineering has become an essential way to make optimal use of the capabilities of LLMs, also in code synthesis. We distinguish between mere prompt engineering techniques and comprehensive frameworks for prompt formatting. Prompt engineering techniques involve adding helpful snippets to a prompt and can always be incorporated into already formatted prompts. On the other hand, full frameworks provide structured guidelines on how to format a prompt effectively. In this study, we highlight the techniques used and their application in code synthesis.

As mentioned previously, White *et al.* [8] 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.* [28] 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[29]. Wang *et al.* [30] 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.*[31] 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 framework in this space is Chain-of-Thought (CoT)[12]. It has been demonstrated that when LLMs generate a chain-of-thought, a series of intermediate reasoning steps, their performance in various tasks significantly improves[12,32,33]. Due to its success, there have been various studies proposing extensions and variants of CoT[34], including ones aimed at code synthesis[35]. Chain of code[36] 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[37] 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-Thought[38] 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.*[39] used this approach to aid in generating Python code for model-based reinforcement learning. A similar approach is Graph-of-Thought (GoT)[40] 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[41,42]. As LLMs have been shown to be especially prone to hallucinations in code synthesis[43,44], prompt engineering techniques designed to mitigate this have also been explored. Our work investigates Chain of Verification (CoVe)[45], which prompts an LLM to have a second look at the responses they give and have them consider whether they made any mistakes. Kouemo Ngassom *et al.*[46] 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, even when using a prompting methodology, a considerable task in itself. Liu *et al.*[47] 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.*[48] employed a different approach for systematic prompt tuning and made use of gradient-guided search.

Lastly, we highlight the study by Ridnik *et al.*[49]. 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 process of solving 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 similar way to GoT. 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 and frameworks, of which there are many more, we refer the reader to Sahoo *et al.* [35].

1.4 Retrieval Augmented Generation

Retrieval Augmented Generation was initially proposed by Lewis *et al.* [50] to help improve LLM-performance in knowledge-intensive NLP-tasks by providing the model with non-parametric memory. The paper describes using a retriever called DPR[51] to retrieve relevant external context and a BART based generator[52] 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. Like 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[53]. Wang *et al.* [54] 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[55] debug the initial code snippet. Parvez *et al.* [56] proposed the REDCODER framework, which utilizes GitHub and StackOverflow 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. ReACC[57] is near identical to this, but solely focuses on code completion. ProCC[58] extends upon 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: encoding lexical semantics of the prompted code; proposing a hypothetical line to add to the prompted code; summarizing 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[59], 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.* [60] 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.* [61], 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 pieces 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.* [62] provide a comprehensive survey on RAG for further reading.

1.5 Finetuning

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

Ogueji *et al.*[63] 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.*[64]. They finetuned `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 an LLM to generate the input for our LLM aimed at code synthesis. Huang *et al.*[65] did the opposite and used a pre-trained LLM to generate labels for unlabeled questions using CoT and CoVe prompting. They then used this generated data to finetune that same LLM. Their results are promising and show state-of-the-art-level reasoning performance on various benchmarks.

Austin *et al.*[66] 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 finetuning dataset was quite small and consisted of 374 code synthesis problems. Nevertheless, they showed that finetuning on a held-out portion of the dataset improved performance by about 10 percentage points across most model sizes. Furthermore, Dong *et al.*[67] also explored finetuning for general code synthesis using the `Code Alpaca` codebase[68]. 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.*[69].

2 Methods

2.1 Choice of DSL

Computational chemistry software packages, such as Gaussian[70], PySCF[71], and ORCA[72], 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 both molecules and materials by describing the behavior of electrons and atoms within molecules. For example, one can use these packages 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, one could argue that the LLMs should be able to perform well or have less to learn, as LLMs designed for code synthesis perform well on Python tasks[7,73]. As for Gaussian, Hocky and White[6] have explored using the `Codex`[7] model to create Python functions to generate Gaussian input files; to emphasize, they did

not directly synthesize 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 in recent times[72]. 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 calculations 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, all while being freely available for personal or academic use. Due to this, and ORCA's solid standing in the computational and theoretical chemistry community, we have opted to create an LLM specialized for ORCA over the other available packages. Like mentioned earlier, to work with ORCA, its user needs to provide it with a syntactically and semantically correct input file

2.2 Dataset Creation

In creation of our datasets used for finetuning, we decided to first generate ORCA input files, and from these input files generate the user prompts due to that this creates a more efficient workflow compared to the reverse. Namely, because we focused on generating executable ORCA input files. If we first created user-prompts and then the corresponding ORCA input files based on those, we would have to check whether they were executable. In case they would not be executable, the user prompt would in turn be invalid as well. By opting for the reverse order we eliminate the risk of generating invalid user prompts.

2.2.1 Input File Generation

For generating an ORCA input file, we considered three types of code segments to generate: keyword lines, input blocks and a coordinate block. Before delving into our three data generation approaches, we will briefly describe these segments and what processing we applied to them across all our data generation methods.

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

Nearly all available keywords can also be called for in input blocks, which provide greater control and customization for users. Namely, they allow for various functionality that is inaccessible through keywords, such as changing the parameters for density functional approximations, defining custom basis sets, and loading settings or coordinates from external files. For every option (like `scf`) various settings are defined, which all have their own range of valid parameters/values. Even though input blocks can be used to do everything keywords can and more, keywords are generally preferred by users in cases of identical functionality for their simplicity, ease of use, and readability. Therefore, we take a keyword-centric approach in our generation methods. Also note that across all generation methods, we add the `pal6` keyword for parallelization over 6 cores before running the file. This was done purely to speed up the calculations, we remove this keyword from the input file after the calculation was performed.

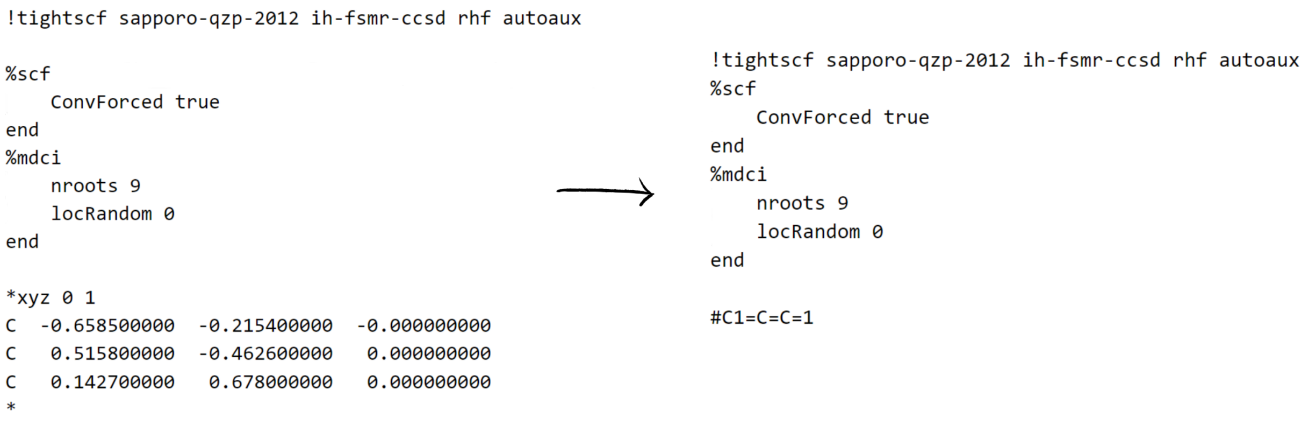


Figure 1: The transformation of a coordinate block (left) to a SMILES molecule (right).

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, by defining the location of basis functions and, ultimately, determining interatomic interactions. 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 a coordinate block. Namely, we append a valid coordinate section of a randomly chosen molecule from our `Gen3Molecules` dataset. After this coordinate block is appended, we run the ORCA input file to see whether it is executable. Only when it is, we save it.

As mentioned in the main materials, we replace the coordinate block in successfully executed input files with a comment containing the SMILES representation of the molecule, as shown in Figure 1. Note that SMILES can contain the '#' character, which is the character used to start a comment line in ORCA. Using this character in the comment line causes an error when running the input file. We therefore replace it with '(hashtag)' to prevent our model from generating faulty code.

The following sections will provide all supplementary details of the three input file generation methods that were presented in the main materials.

2.2.1.1 Brute-force

In the brute-force approach, we randomly combine keywords, options, and settings as follows:

1. For the keyword line, we select a sample of up to ten[1] keywords from those defined in the ORCA manual.

2. We randomly decide whether to add an input-block to the input file. If we do, we randomly select an option, and then randomly choose up to five[1] settings defined for that option.

[1]The choices of the maximum keywords and settings were quite arbitrary, but motivated by the fact that, in experimentation with the brute-force method, it nearly never occured that a functional input file with over ten keywords or five settings was generated

3. We append a random molecule and attempt to run the generated input file with ORCA. If the input file finishes the simulation without any errors, we replace the coordinates block with the corresponding SMILES and save it to the dataset.

This method can be quite slow in generating valid ORCA input files. Additionally, combining elements without prior knowledge of their compatibility can result in numerous invalid files and even if these files are runnable, they may not accurately reflect real-world input files. However, this approach is easy to implement and highly generalizable to DSLs. Note that the options and keywords were scraped from chapter 6 of the ORCA manual.

2.2.1.2 Manual-based

In the manual-based approach, we leverage the fact that the ORCA manual provides many examples of how one is to perform specific calculations. These examples include code blocks with either full input files or input blocks/keywords on their own. We extracted all unique instances of individual keyword lines and input blocks from these examples. We then used these sections as follows:

1. We randomly choose a keyword line from the extracted ones.
2. We randomly decide whether to add an input block to the input file. If we do, we randomly select an input block out of the extracted ones.
3. We append a random molecule and attempt to run the generated input file with ORCA. If the input file finishes the simulation without any errors, we replace the coordinates block with the corresponding SMILES and save it to the dataset.

This method relies on the intuition that the individual sections extracted from the manual contain logical combinations of keywords in the keyword line and logical combinations of options and settings in the input block. Consequently, generating input files should be somewhat quicker when compared to the brute-force approach, and the generated input files are more likely to correspond to real-world data. However, this method requires the DSL in question to have enough code snippets available from which meaningful sections of code can be extracted.

2.2.1.3 Rule-based

In the rule-based approach, we utilize prior knowledge of how to construct correct input files to ensure that every generated input file is valid. By applying predefined rules and structures based on expert knowledge and documentation, we can systematically create input files that adhere to the required syntax and semantic rules of ORCA.

Due to time constraints, we narrowed the scope of calculation types to be generated based on rules. We selected the ORCA calculations most frequently used, according to internal domain expertise. The implemented calculations are shown in Figure 2. Note that we decided to refrain from implementing resource management in these calculations, such as the requesting cores and memory, Namely, because the use of these settings is very machine-dependent.

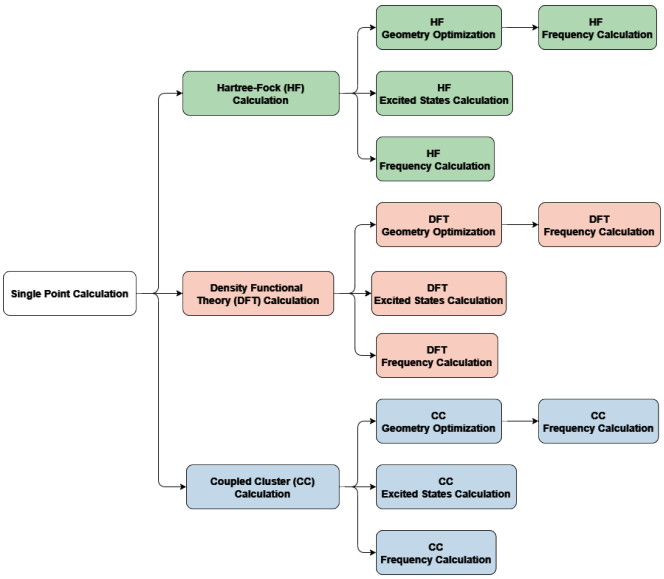


Figure 2: A simple overview of the implemented calculation types. The arrows indicate the inheritance of functionality from the parent class. Solvation is not shown here, but is considered a seperate calculation type in our generation method.

The implemented calculations adhere to rules derived from the descriptions and guidelines provided in Chapters 8 and 9 of the ORCA manual. Additionally, by experimentally running the generated files and refining the rules based on encountered warnings, we developed a system where none of the generated input files result in any warnings. The final system and the rules for each calculation can be found in the code repository at https://git.lwp.rug.nl/pollice-research-group/ORCAInputFileSynthesis. We continue to provide a brief overview of the essential components in the calculations. Note that this overview will go on to freely use various terms from computational chemistry and the following is therefore mainly aimed at readers familiar with this field of research.

Each calculation starts by selecting an appropriate basis set, given the randomly chosen molecule. We check which basis sets are available for the molecule based on its elements and randomly choose from the compatible options. Appropriate auxiliary basis sets are appended to the input file when necessary based on the methods that were selected.

Aside from the basis set, the Hartree-Forck (HF) type also depends on the molecule. For radicals, with their unpaired electrons, an unrestricted or restricted open HF approach is necessary. Conversely, for molecules where all electrons are paired (i.e., closed-shell), the Restricted HF method (RHF) applies. While ORCA defaults to RHF, we choose to always verbalize the HF-type in the keyword-line, that is also in the case of molecules without radical electrons. In doing so, we hope to teach the LLM to use the correct HF type for a given molecule.

Moreover, across all simulation types, we allow for a continuum solvation calculation to be added. Solvation effects are critical for accurately modeling the behavior of molecules in different environments. When solvation is desired, we use the conductor-like polarizable continuum model, or `CPCM`, with a randomly chosen solvent.

After these core parts of the input file are defined, we move on to generate the parts that are inherent

to the different calculation types. Below, we list the main functionality and options for all calculations. Note that we also utilize other functionalities of the ORCA software within these options, which can also be used in a standalone manner (e.g. Møller–Plesset perturbation theory).

- For **HF single points**, we allow for the user to specify whether to use a Møller–Plesset perturbation theory (MP2) method and an RI approximation with the corresponding input blocks.
- For **Density functional Theory (DFT) single points**, we randomly choose the density functional from all the ones that are implemented in ORCA and always use tight SCF convergence. Moreover, we allow for RI approximations, the non-local correction, and dispersion corrections to be used. Lastly, we randomly choose a grid type to use for the calculation.
- For **Coupled Cluster (CC) single points**, we randomly choose a CC level from all ones that are implemented in ORCA and allow the use of `mdci` density calculations.
- For **geometry optimizations**, we randomly select an optimization type (we also allow for saddle point optimization) together with a geometry convergence type. We allow for both relaxed scans and (re)calculation of the Hessian. Moreover, we allow for constraining bond lengths and angles.
- For **excited states calculations**, we allow for various settings in the `tddft` or `cis` input blocks to be applied (which are used for DFT and HF calculations, respectively). Namely: simulation of triplets, the printing of the population analysis, using spin flip calculations, using non-adiabatic couplings, inclusion of spin-orbit coupling, use of spin component scaling, use of double correction, and different modes of `tddft` calculations. By default, we use nine excited states or roots to calculate as this was the number of roots most often used in the ORCA manual. For the CC excited states calculations, we allow for triplets and QROs to be used. Moreover, one can choose to filter out states with doubles excitation character. For DFT, we randomly choose a hybrid or double-hybrid functional and for CC we use a random `mdci` method for excited states calculations (e.g. `STEOM-CCSD`
- For **frequency calculations**, we provide the option of whether or not to also perform a geometry optimization. If this optimization is performed, we always use tight optimization convergence. Moreover, we allow for use of central differences to approximate the gradient and Hessian numerically.

Because we aim to avoid assumptions about the prevalence of different data types, this method generates an equal amount of input files for each calculation type. Additionally, to ensure a diverse range of configurations, we randomly decide whether to include specific settings (described above) for each calculation when generating an input file. The exception to this is the use of solvation. We treat solvation as a separate calculation type in our generation scheme. For each solvation calculation, we randomly select a base calculation type and then apply the solvation model to it.

This method guarantees the generation of valid input files, eliminating the trial-and-error aspect and increasing efficiency compared to previously described methods. Additionally, the input files have a structure that presumably resembles real-world data most closely across all three generation methods.

Implementing this method, however, is more challenging than our brute-force or manual-based approach, as it requires significant prior knowledge and effort to establish the rules, which are also laborious

to implement. Moreover, these rules are specifically tailored to the DSL in question, making the approach less generalizable to other DSLs when compared to the other two approaches.

2.2.1.4 Dataset Statistics

We were interested in comparing the complexity of the computer-generated input files against the real-world dataset. The corresponding comparison based on the number of keywords, the number of input blocks, and the number of setting lines is illustrated in Figures 3-5. The corresponding quartiles are provided in Table 1. These results demonstrate that the real-world dataset contains input files with significantly higher complexity. The dataset created via the rule-based approach comes closest in terms of the number of keywords, input blocks, and setting lines. The datasets created via the brute-force and the manual-based approaches show similar complexity, with the brute-force approach resulting in input files with slightly higher complexity.

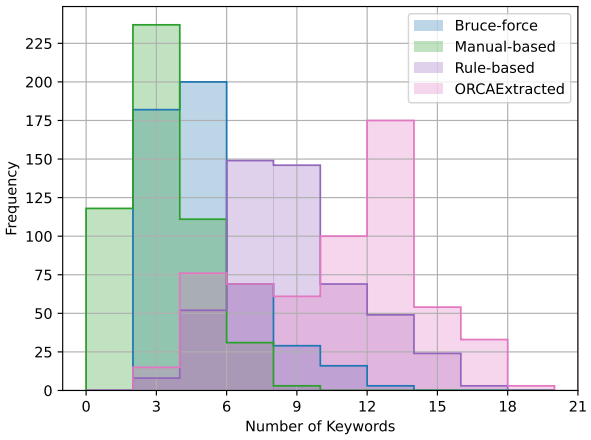


Figure 3: Unnormalized histograms of the number of keywords for datasets of this work.

Dataset	Keywords			Input Blocks			Setting Lines		
	Q1	Q2	Q3	Q1	Q2	Q3	Q1	Q2	Q3
Brute-force	3	4	5	0	0	1	0	0	2
Manual-based	2	3	4	0	0	1	0	0	1
Rule-based	6	8	10	1	2	3	2	3	5
ORCAExtracted	7	11	13	2	3	4	3	5	9

Table 1: Quantiles for the number of keywords, the number of input blocks, and the number of setting lines in both the computer-generated datasets and the real-world dataset. Abbreviations: Q1 = first quartile, Q2 = median, Q3 = third quartile.

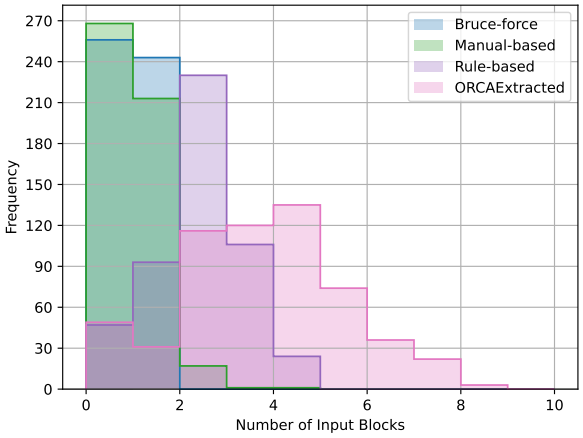


Figure 4: Unnormalized histograms of the number of input blocks for datasets of this work.

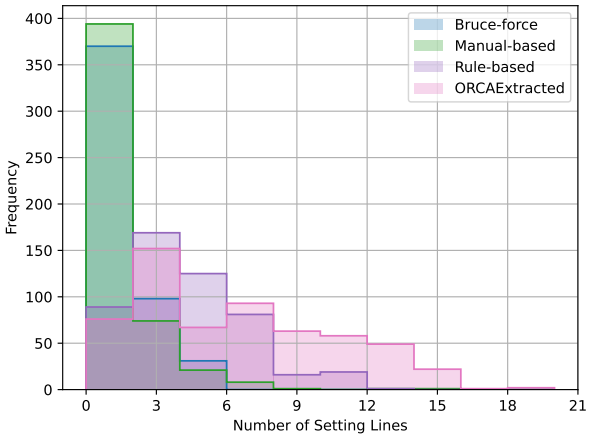


Figure 5: Unnormalized histograms of the number of setting lines for datasets of this work.

2.2.2 User Prompt Generation

The exact system prompt used for the transformation from a crude prompt to the final prompt is a simple prompt explaining the format of the crude input file description that instructs the model to transform it into a description of a desired chemistry simulation similar to how a user might write it, presumably. We also instruct the model to write the synthetic prompt between `!PROMPT!` tags, to make extraction more straightforward. An example of a final synthetic prompt, which was created by GPT-4o, is given in Figure 6b. The full prompt is as follows:

```
**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.
```

An additional example, beyond the one provided in the main materials, illustrating the transformation from a crude to a final prompt is shown in Figure 6.

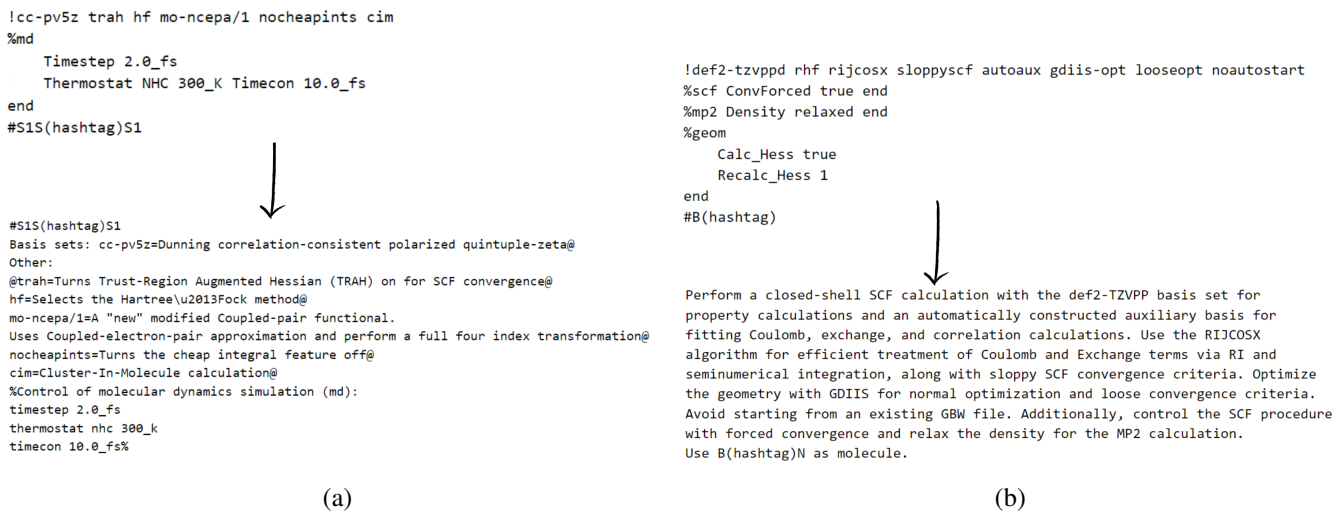


Figure 6: The transformation of an input file to a crude description of it (a) and the transformation of an input file to the final synthetic prompt describing it (b).

2.3 Model Architecture

2.3.1 Inference

Given a system prompt (S), an input/user prompt (X), we have our LLM (m) produce output o. This is described more formally in Equation 1.

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

Presumably, o contains an ORCA input file $\hat{y}$, which we try to extract in two ways. In the first approach, we determine if the LLM generated a code block, meaning it enclosed text in three opening speech marks (‘). If this is the case, we extract the text inside this code block. If this approach fails, we move on to performs a regex search based on the structure of an ORCA input file. If both of these methods fail, we use o as $\hat{y}$.

2.3.2 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 finetuned (Section 2.3.4) on the synthetic datasets (Section 2.2) and the model that has to interpret and use context retrieved by RAG (Section 2.3.5). The model we use in this study is `gpt-3.5-turbo-0125`.

`GPT-3.5 Turbo` is an updated version of `GPT-3`[31] that uses different training paradigms to get to improved performance. Note that the full technical details of its training process are not made public[74]. 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.

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. We opted for using the `GPT-3.5-turbo-0125` model as it is, at the time 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[75]. We opted to use a general-purpose LLM, and not a specialized coding model like `CodeT5`, as we figured that the general-purpose 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 4. Also note that, for 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 potential LLMs, particularly `GPT-4(o)` , `Gemini 1.5`[76], `Claude 3`[77], `DeepseekV2`[78] and `LLama 3`[79].

2.3.3 Prompt Engineering

We apply Prompt engineering in the system prompt, which 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 cannot be overridden by the user and is utilized 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 2.2.2.

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. However, they differ in their approach of how they instruct the LLM to reason. The prompts are aimed at ensuring that the LLM reasons step-by-step throughout the construction of the different parts of an ORCA input file (described in Section 2.2.1.

Specifically, the prompts guide the LLM through the following steps:

1. Identifying the relevant ORCA keywords.
2. Determining the necessary input block options.
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.

The six system prompts we created can be found in Appendix A.

2.3.4 Finetuning

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 finetuning, we used 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 the 'Basic' prompt engineering technique, we defined the desired ORCA input file as the desired response, or label, in the finetuning task. However, when using prompt engineering, it was 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
```

2.3.5 Retrieval Augmented Generation

In this study, we used a naive-RAG scheme much like the documentation-based RAG by Zhou *et al.*[61]. Naive-RAG is the most basic RAG methodology[62]. Due to its simplicity and explainability, it was deemed an appropriate initial approach for this research.

During initial experimentation with RAG, we found that the model did not indicate how it used context, often forgot to use context or used 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 to be mindful only to use the context when it is actually helpful or relates to the users request. The full prompt is shown below:

```
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.
```

2.4 Experimental Setup[2]

2.4.1 Data

We generated three training datasets for finetuning (Section 2.2), using the described generation-methods. To reiterate, these are referred to as brute-force, manual-based and rule-based. For each of these, we generated 500 supervised entries to use during finetuning (Section 2.3.4).

For testing and validation of our experiments (Section 2.4.2), we used another dataset which we will refer to as `ORCAExtracted`. As explained earlier, we decided that it was essential to use real-world data for the evaluation sets as to be able to accurately assess the performance of our models. Generally, one of the most straightforward ways to extract such data for a DSL is to extract it from its documentation. However, scraping input files from the ORCA manual was deemed problematic, because this documentation also gets used in generating the training data. Therefore, we used two other sources: ioChem-bd.com and internal data from the Pollice Research Group.

IoChem[80] is a computational chemistry repository and platform that provides access to a wide range of chemical data from various types of calculations, such as electronic structure calculations. It is designed to facilitate the sharing of computational data within the scientific community, including data from

[2]All datasets we created and the code used to run the described experiments can be found on https://git.lwp.rug.nl/pollice-research-group/ORCAInputFileSynthesis

universities. Consequently, it houses many calculations performed by computational chemistry software packages like Gaussian and ORCA, and includes the input files used to run these calculations.

We filtered the search section based on calculations that used the ORCA software and extracted 71,735 ORCA input files. This seems like a significant amount of useable data, but many research groups perform similar calculations for different molecules or adjust a single parameter (like memory) in an input file across calculations. Therefore, we filtered out duplicate files by retaining only those with unique keywords and options, resulting in a total of 422 unique input files. The same process was applied to 241 input files gathered from chemistry experts in the Pollice Research Group.

Some of the extracted input files were written for ORCA version 4 and thus used outdated language (e.g., using the `grid` keyword instead of `defgrid`). All outdated language was replaced with corresponding new terms, or if no corresponding terms existed it was simply omitted. To check the validity of these input files, we executed them and saved only those with valid execution. This left us with a total of 588 input files, ready for post-processing.

Our post-processing of this data involved several steps to ensure a fair evaluation in relation to our training data. Firstly, we removed all comment lines, as our training data did not contain those (except for the SMILES). The molecules in the extracted data were generally much larger than three atoms and were thus not encountered in our training data. Therefore, we replaced the coordinate block of the extracted input files with a random molecule from `Gen3Molecules` (Section 2.2.1) that was valid for the basis set used. Additionally, we removed both empty newlines from the input files and tabs from input blocks to both limit the number of tokens in the final LLM output and standardize the indentation format of all input files (indentation does not affect ORCA functionality). Finally, to standardize capitalization, we converted all text in the extracted input files to lowercase, consistent with our approach adopted for generating input files.

The final step to complete our test and validation sets was to gather prompts for the input files we extracted. Unfortunately, the extracted data was generally undocumented and we thus had no means of extracting prompts together with the corresponding input files. Therefore, we created prompts for the extracted input files with the approach described in Section 2.2.2.

Table 2 provides an overview of the amount of unique terms and skewness in the generated datasets and the validation and test splits of the `ORCAExtracted` dataset. Skewness is a measure of the asymmetry of the probability distribution of a real-valued random variable about its mean. Positive skewness indicates a distribution with an asymmetric tail extending towards more positive values. Equation 4 shows its mathematical definition, where, in our case, n is the number of unique keywords/options, x_i an individual keyword/option count, $\bar{x}$ represents mean count over all options/keywords, and s the standard deviation.

$$\text{Skewness} = \frac{n}{(n-1)(n-2)} \sum_{i=1}^{n} \left(\frac{x_i - \bar{x}}{s} \right)^3 \tag{4}$$

The brute-force dataset exhibits the lowest skewness for both keywords and options, indicating a more symmetric distribution. In contrast, the rule-nased dataset shows high keyword skewness, suggesting significant asymmetry in its distribution, along with a relatively small number of options compared to other datasets. The manual-based dataset has fewer unique keywords and higher skewness, reflecting considerable asymmetry. The validation and test sets also have relatively few unique keywords but display less skewness compared to the Manual-Based dataset. For a more detailed overview of the frequencies of the exact options and keywords, we refer the reader to Appendix B.

Table 2: Overview of the variability and skewness of the generated datasets and the validation and test split of the `ORCAExtracted` dataset. The highest variability and the lowest skewness across the datasets is in bold.

Dataset	Unique Keywords	Unique Options	Keyword Skewness	Option Skewness
Brute-force	**530**	25	**1.279**	**1.040**
Manual-based	165	**31**	4.681	2.687
Rule-based	336	9	8.516	1.540
Validation	153	27	2.649	2.092
Test	159	26	2.658	1.932

Lastly, we had chemistry experts from the Pollice Research group write three prompts that they would put into a system meant for generating ORCA input files. This data was held out for a separate testing set which we will refer to as `RealPrompts`. We decided to use this separate dataset because our evaluation set consists of synthetically created user prompts. We wanted to test whether our finetuned models could respond appropriately to real user prompts. The chemistry experts were asked to write a prompt for a ChatGPT-like model, if they would want it to create an ORCA input file. The goal here was to get them to create a prompt as close to what a user of our model would provide.

2.4.2 Experiments

The goal of our experiments was to explore how prompt engineering techniques, retrieval-augmented generation (RAG), and finetuning impact the performance of a Large Language Model (LLM), specifically GPT-3.5-turbo, in generating accurate and functional input files for ORCA, a chemistry simulation domain-specific language (DSL). This goal was divided into the following sub-questions, where performance was measured using the metrics described in main text:

1. How does finetuning on 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?
3. How does RAG affect performance?
4. What are the interaction effects between the three techniques, if there are any?

After evaluation of all these configurations, a final experiment was conducted on the `RealPrompts` dataset, as described in Section 2.4.1. We used the model that came out with the best $F1_{avg}$ to predict input files from these prompts in hopes of being able to provide a qualitative assessment on how our model performs on real-world prompts.

2.4.3 Hyperparameters

In tuning the available hyperparameters, we distinguish between those used during finetuning, thus relating to what model we end up creating, and those where one simply alters the parameters when running the model. The latter has only relatively few options, as OpenAI does not allow for much configuration of their models in their API endpoints. We kept the temperature at the default, namely 1. This balances creativity and coherence in responses. Moreover, for all model configurations, we ran RAG with values of $K = \{1, 5, 10, \textit{Limit}\}$ and chose the value where our model got to the highest validation accuracy for final evaluation on the test set.

As for the finetuned models, a grid search was performed for every model configuration. This was deemed necassary as the different finetuned models all are trained within a different training landscape. Namely, because the models are training on different datasets and tasks (considering the different prompt engineering techniques used). The grid search involved the following options, where η multiplier refers to the multiplier of the η used during pre-training:

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

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 $F1_{avg}$-score for further evaluation. This mismatch between loss function and evaluation metric will be further discussed in Section 4. The final hyperparameters for the different models are shown in Table 3.

Table 3: The used hyperparameters for all models, including the base models that do not use finetuning. When models use CoT together with finetuning, the models were trained with synthetic CoT-labels.

Model	CoT	Finetuning	Epochs	Batch Size	η multiplier	K
GPT-3.5 Turbo	✗	✗	✗	✗	✗	10
GPT-3.5 Turbo	✓	✗	✗	✗	✗	10
GPT-3.5 Turbo	✗	Brute-force	5	1	2	1
GPT-3.5 Turbo	✓	Brute-force	8	1	2	5
GPT-3.5 Turbo	✗	Manual-based	3	1	2	1
GPT-3.5 Turbo	✓	Manual-based	8	1	2	1
GPT-3.5 Turbo	✗	Rule-based	3	1	2	1
GPT-3.5 Turbo	✓	Rule-based	5	1	2	1
GPT-4o	✗	✗	✗	✗	✗	3
GPT-4o	✓	✗	✗	✗	✗	5
GPT-4o	✓	Manual-based	3	1	2	1

3 Supplementary Results

This section presents the results that were left out of the main materials. These include the ORCA errors that were encountered after trying to execute synthesised inputs and the evaluation of our best model when used with real-world user prompts.

3.1 Hyperparameter Optimization

We provide a concrete example of the found mismatch between $F1_{avg}$ score and validation loss in Table 4, which shows how the loss and $F1_{avg}$ score differ over a changing amount of epochs. We observe that the loss gets higher the more epochs we train the model for, but that this is clearly not the case for the $F1_{avg}$ score.

Table 4: A sample of the results from our grid search, displaying both validation loss and validation $F1_{avg}$ of the brute-force model without CoT, over different epochs for an η multiplie of 2 and a batch size of 1. The best performance across both metrics is bolded.

# Epochs	**Loss**	**$F1_{avg}$**
1	**2.402**	0.212
3	2.666	0.287
5	3.012	**0.293**
8	3.332	0.292
10	3.812	0.287

3.2 Base Models

Figure 7 shows the different error types ORCA returned after trying to execute the invalid files that were generated by the models.

Across all base-model configurations, we observe that the two most common errors encountered are the use of an unknown identifier (the first term in a setting line) and the use of an unrecognized keyword. When utilizing RAG, we find that the model more frequently encounters the unknown identifier error, but less often generates non-existent keywords or symbols (the second term in a setting line). Another notable observation is that the base-model employing both CoT and RAG is the only model that encounters errors where an ORCA section identifier (!, %, or *) is not generated correctly.

Table 5 presents further analysis of the unrecognized keywords generated by different models. Note that the Levenshtein distance[81] is a metric for measuring the difference between two strings by counting the minimum number of single-character edits (insertions, deletions, or substitutions) required to transform one string into the other. We find that models using CoT tend to generate entries that are closer to real keywords, on average, when compared to other models. Moreover, the percentage of unrecognized keywords generated out of all keywords generated is lowest for the model using both RAG and CoT.

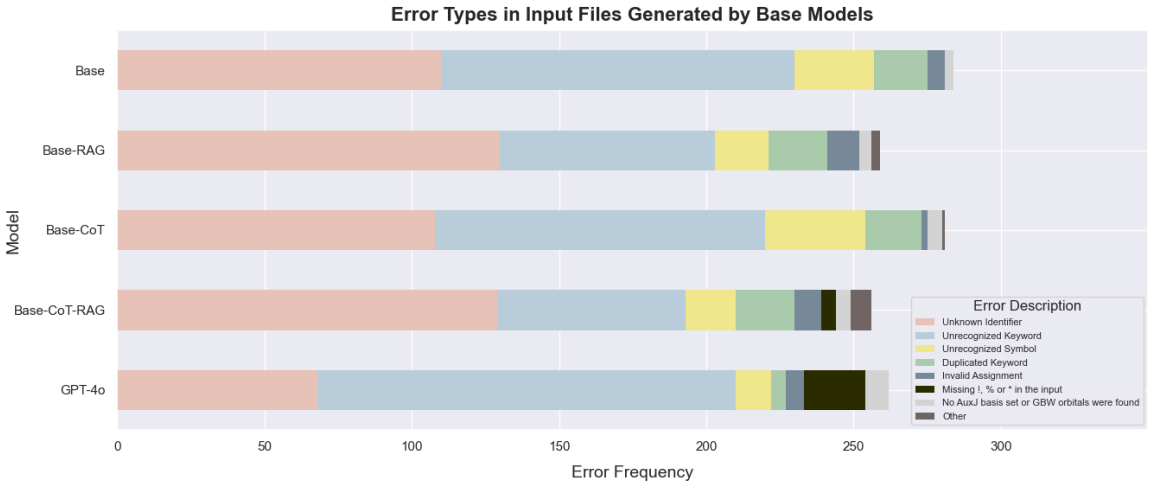


Figure 7: The different error types encountered when running the input files generated by the respective base models. An unknown identifier and unknown symbol mean that the first and second term of a setting line are unknown to ORCA, respectively. An invalid assignment refers to the fact that the second term of a setting line is known to ORCA, but that it cannot be used in this context. Errors were grouped in the 'other' category if they were not encountered five or more times by any of the models. Note that the ORCA compiler terminates after encountering an error immediately.

Table 5: The average Levenshtein distances of unrecognized keywords to their closest ORCA keyword, along with the percentage of unrecognized keywords encountered in the keyword line(s) where at least one of the generated keywords was erroneous, for the base-model configurations with the last row being the `GPT-4o` baseline. For both metrics, the lowest value is in bold.

CoT	RAG	Levenshtein Distance	% Unrecognised Keywords
✗	✗	4.332	43.160
✗	✓	4.024	39.029
✓	✗	**3.787**	39.245
✓	✓	3.975	**34.593**
✗	✗	4.275	28.930

3.3 Finetuned Models

The different error types across all model configurations are presented in Figure 8.

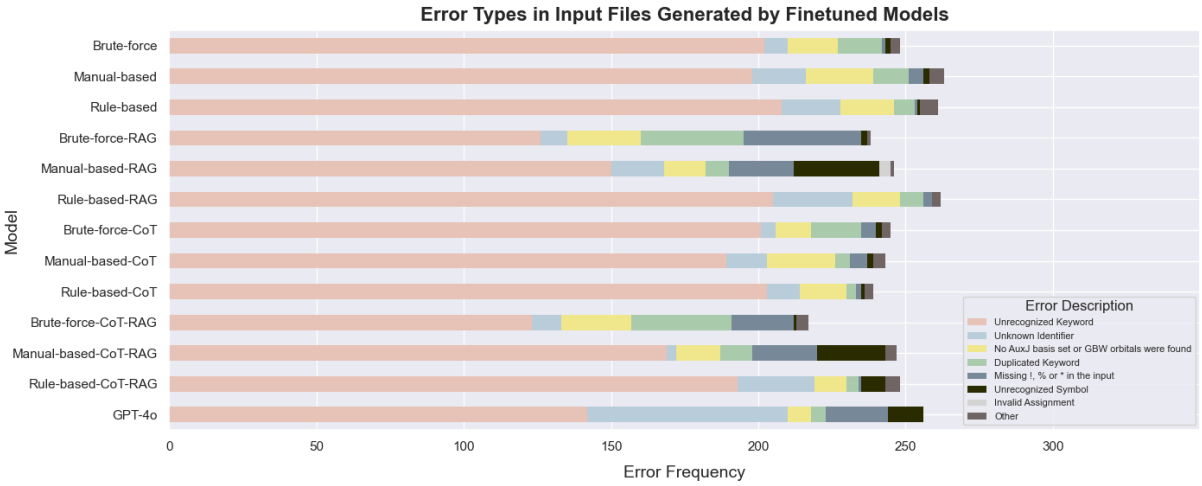


Figure 8: The different error types encountered when running the input files generated by the respective finetuned models. An unknown identifier and unknown symbol mean that the first and second term of a setting line are unknown to ORCA, respectively. An invalid assignment refers to the fact that the second term of a setting line is known to ORCA, but that it cannot be used in this context. Errors were grouped in the 'other' category if they were not encountered five or more times by any of the models. Note that the ORCA compiler terminates after encountering an error immediately.

Compared to the baseline models, we observe that the most frequent error encountered remains the presence of unrecognized keywords in the input files. This error occurs even more frequently when compared to the baseline models. However, we also note a significant decrease in unknown identifier errors, which is in line with improved $F1_{setting}$ performance.

As for distinctions between the finetuned models themselves, we observe that rule-based models exhibit a relatively high frequency of unknown identifier errors. Additionally, manual-based models utilizing RAG encounter numerous unrecognized symbol errors, a trend less prevalent in other models. Moreover, both brute-force and manual-based models employing RAG tend to produce a notable quantity of input files with duplicate keywords.

A more detailed analysis of the most common error (unrecognized keywords) is presented in Table 6. Generally, compared to baseline models, finetuned models generate fewer non-existent keywords, and those generated are closer on average to existing keywords. The exception to this trend is observed with the brute-force model using RAG, where performance in this regard is notably worse.

Across the finetuned models, the rule-based model consistently generates the fewest unrecognized keywords relative to the total number of keywords generated. Manual-based models generally produce unrecognized keywords that are closest to real ones compared to other finetuning types, except when both CoT and RAG are employed. Notably, brute-force models perform less effectively in this error type, displaying higher Levenshtein distances and greater quantities of unrecognized keywords. We find that RAG does not enhance performance within this error category, except marginal improvements for the rule-based dataset. Conversely, the use of CoT reduces Levenshtein distances for both brute-force and rule-based models, and decreases the occurrence of unrecognized keywords across all models. However, combining RAG with CoT leads to inferior performance compared to using CoT alone.

Table 6: The average Levenshtein distances over unrecognized keywords to the closest ORCA keyword. The last row represents the results gathered from `GPT-4o`.

CoT	RAG	Finetuning	Levenshtein Distance	% Unrecognised Keywords
✗	✗	Brute-force	3.681	34.352
✗	✗	Manual-based	**2.684**	34.513
✗	✗	Rule-based	3.249	29.799
✗	✓	Brute-force	4.421	37.392
✗	✓	Manual-based	2.702	30.080
✗	✓	Rule-based	3.008	25.806
✓	✗	Brute-force	3.478	31.862
✓	✗	Manual-based	2.889	27.452
✓	✗	Rule-based	2.911	**25.034**
✓	✓	Brute-force	3.704	38.323
✓	✓	Manual-based	3.046	32.829
✓	✓	Rule-based	2.834	25.499
✗	✗	✗	4.275	28.930

3.4 Real-world User Prompts

Figures 9, 10, and 11 present the results of running both the unaltered `GPT-3.5 Turbo` (using the 'basic' system prompt) and the `Manual-based-CoT` models on the `RealPrompts` dataset described in Section 2.4.1. Note that prompts from `RealPrompts` did not specify what molecule was to be used, hence we ignore the predicted SMILES of the models in the presentation of our results.

Firstly, Figure 9 displays the output of both models for a requested `CASSCF` calculation. `GPT-3.5T urbo` correctly uses the specified basis set but assigns the keyword for initiating a `CASSCF` calculation incorrectly. It translates the request for the number of orbitals and electrons used into a non-existent keyword instead of accurately placing it in an input-block for `casscf`. Additionally, it includes a maximum iteration specification for `SCF`, which, although valid, was not requested. `Manual-based-CoT` manages to predict both keywords correctly but creates an input block that does not correspond to the user's request. Moreover, it generates `EXITONS` within a `TDDFT`, which is not a valid identifier and was likely intended to be `EXCITATIONS`.

Using CASSCF (6 active orbitals and 6 electrons) and def2-QZVP basis set, please perform a single point energy calculation.

```
!casscf(6,6) def2-qzvp
%scf
  maxiter 1000
end
#C
```

I used the following step-by-step logic to get to my input file prediction:
Step 1: The basis sets used are def2-qzvp.
No density functional is used. Further keywords used are: casscf
Step 2: The input block options that should be used are: tddft
Step 3: N/A
Step 4: The desired SMILES is B(hashtag)P
Step 5: Given the answers to the above steps, the final ORCA input file is:

```
!def2-qzvp casscf
%TDDFT SGRADLIST 0,1,2
EXCITONS 1
END
#B(hashtag)P
```

Figure 9: User prompt requesting a `CASSCF` calculation (left) with unaltered outputs of `GPT-3.5 Turbo` (middle) and `Manual-based-CoT` (right).

Secondly, Figure 10 illustrates the generated input files for both models when the user requested a DFT-based frequency calculation. `GPT-3.5 Turbo` produces an extensive output with various invalid input blocks, except for the correct `SCF` block. The model did not recognize that many of the requested simulations (i.e., Hirshfield population analysis, frequency calculation, and RI approximation) are available in keywords. It, however, accurately predicts both the basis set and the density functional. For `Manual-based-CoT`, these are also correctly generated. Moreover, it accurately predicts `anfreq` and `hirsh` for performing the requested analytical frequency calculation and Hirshfeld population analysis, respectively. However, it incorrectly identifies `nmr`, `maxcore`, and `maxiter` as keywords instead of placing them in input blocks. Additionally, it confuses `maxcore` for a parameter to be used to parallelize the calculation, while the correct identifier for this is `nprocs`.

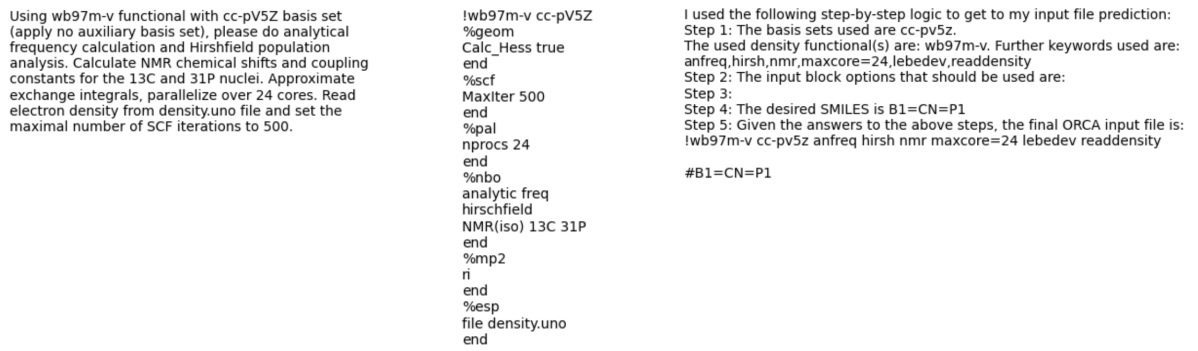


Figure 10: User prompt requesting a DFT-based frequency calculation (left) with unaltered outputs of `GPT-3.5 Turbo` (middle) and `Manual-based-CoT` (right).

Lastly, Figure 11 presents the results of requesting a local energy decomposition analysis. `GPT-3.5Turbo` uses non-existent keywords `dlpno-eda` and `ccsd(t)`, mixing up the user-requested keywords. Moreover, it makes a mistake in trying to use very tight SCF convergence in that it uses the invalid identifier `scfconv` instead of the appropriate `Convergence` with the assignment `VeryTight` to achieve this. `Manual-based-CoT` correctly predicts `dlpno-ccsd(t)` along with `verytightscf` for stringent SCF convergence. It however mistakes `eda` for being a keyword as it is an identifier within `scf` input blocks. Unfortunately, neither model recognizes the need for the `led` keyword to implement local energy decomposition analysis.

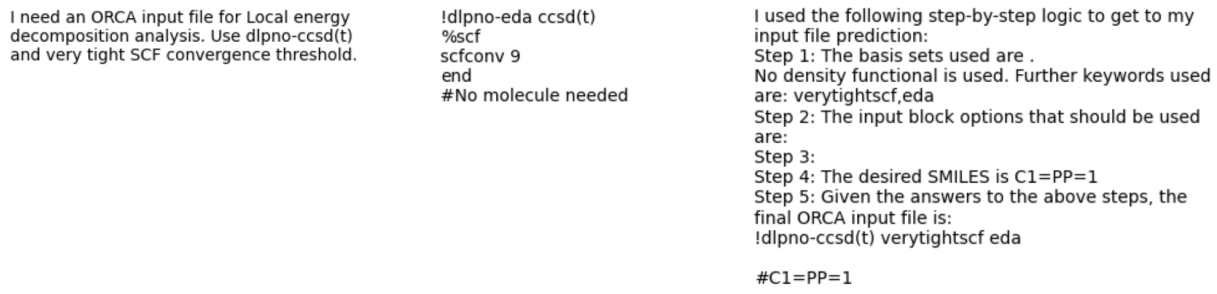


Figure 11: User prompt requesting local energy decomposition analysis (left) with unaltered outputs of `GPT-3.5 Turbo` (middle) and `Manual-based-CoT` (right)

4 Discussion

The goal of this research was to find out how prompt engineering techniques, Retrieval-Augmented Generation (RAG), and finetuning impact the ability of a Large Language Model (LLM) to generate accurate and functional input files for ORCA chemistry simulations. To achieve this, we proposed three ways of generating a training dataset for a Domain Specific Language (DSL) and measured the results of running `GPT-3.5 Turbo` in various configurations, utilizing finetuning, RAG and prompt engineering, on a dataset of real-world ORCA input files called `ORCAExtracted`. Moreover, to further investigate real-world performance, we ran both our most basic and best performing models on the `RealPrompts` dataset. This section goes on to discuss the ORCA errors encountered when executing the different models synthesized input files and the `RealPrompts` experiment.

4.1 ORCA Error Analysis

In the main materials, we observed that the $F1_{keywords}$ was relatively high compared to $F1_{options}$ and $F1_{setings}$. However, we saw in the presented ORCA error analysis that the majority of the errors encountered in running the generated files are due to the model generating non-existent keywords, and that, in these erroneous files, close to half of the keywords were unrecognisable to ORCA. Therefore, despite this high value, along with the fact that the F1 score for keyword synthesis is low in an absolute sense, the model still has significant room for improvement in this area as well.

Additionally, the notion presented in the main materials that additional context from RAG helped the model reduce the number of erroneous keywords is also supported by the fewer predicted erroneous keywords and, when errors did occur, they were closer to existing ones compared to the baseline performance. However, there was an increase in unknown identifiers, possibly because the compiler processed further into the input file before encountering errors. Previously, the compiler probably stopped at the keyword line, withholding it from seeing any subsequent errors when the baseline model was used. Notably, we also observed that using CoT makes the unrecognised keywords the model generates closer to existing ones. While the F1 scores did not increase significantly, this does indicate that the CoT prompting increases model performance.

For finetuned models, we observed that models utilizing RAG start to generate input files where section identifiers, the most basic syntactical parts of an ORCA input file, are forgotten. This further strengthens our point about the finetuned models 'forgetting' how to handle external context, or getting 'confused' by it.

Moreover, we also observed more indications that the brute-force approach did not result in the models understanding keyword synthesis in a meaningful way. In particular, the outputs of the model had the highest amount of unrecognised keywords in erroneous files and these keywords were also the furthest from correct ones across all the different finetuned models.

The worse than expected performance of the rule-based approach could be a result of the comparably high keyword skewness. This indicates that a few keywords take up a majority of the encountered keywords, which could cause the model not to learn enough about the different available keywords. This skewness was the least for options, and there we saw that the rule-based model did achieve comparably good performance.

4.2 Real-world User Prompts

When comparing our most basic model (`GPT-3.5Turbo`) with our best-performing model (`Manual-based-CoT`) on `RealPrompts`, we observed that the performance of our finetuned model seemed to generalize well to prompts different from those it was trained on. The `Manual-based-CoT` model consistently outperformed `GPT-3.5 Turbo` in accurately predicting domain-specific keywords and parameters. However, both models struggled with accurately structuring input blocks according to user prompts, often misplacing or misinterpreting keywords and identifiers. This indicates that while the finetuned model improves keyword prediction, it still lacks robust understanding of appropriate options and settings placement. Similar challenges were observed in the F1 scores for these specific sections (cf. main materials) when applying `Manual-based-CoT` to the `ORCAExtracted` test set, as expected. This supports the notion that our approach to generating user prompts and training the model correlates with real-world performance. It is essential to note that this assessment is qualitative in that only three prompts were evaluated, and further research is necessary to delve deeper into these findings.

4.3 Limitations

Regrettably, there were some flaws in the methodology which could have had an impact on the generalizability and the reliability of the results.

The first issue is that all results were obtained from a single run, which makes them susceptible to variation. In generating code/text with of LLMs, randomness plays a significant role, and our results could have changed considerably if averaged over multiple runs. Additionally, we saw that the highest value for the η multiplier in our grid search was selected across all finetuned models. It is possible that a higher learning rate could have improved performance. However, due to our initial choice of search space for η, we may have missed out on identifying these better-performing models.

Like mentioned previously, we did not find the time to implement a rule-based scheme for all possible ORCA calculations. This could have resulted in the rule-based models not learning about calculations that were prevalent in the `ORCAExtracted` test set thus decrementing the finetuning performance. It might be, that if we implemented all possible calculations, the rule-based models would have ended up achieving the best performance. Additionally, we did not optimally take advantage of the fact that we know the type of calculation associated with every generated input file for this dataset. We could have made use of this by adding the calculation type to the prompt, as to have the LLM learn more about the types of calculations.

Moreover, our approach to generating user prompts precluded us from utilizing the documentation for RAG that contains detailed tables describing keywords and their options. This documentation could have provided the best context for synthesizing input files. As a result, there is a possibility of misrepresenting how RAG could be effectively applied in a real-world setting, where such contextual information is typically used.

Of course it is also worth mentioning that our quantitative evaluation cannot be compared to real-world performance, as the user prompts were synthetically created. Although they seem to correspond well to what a user might request, there is really no telling. Our experiment on the `RealPrompts` dataset tries to compensate for this, but is very limited in the amount of data that is used.

Lastly, the error types our models encountered were generally because of the fact that unrecognised

keywords were used. However, this does not mean that there were no other errors as the ORCA compiler simply stops at the first error it encounters. Therefore, we did not provide a full overview of the erroneous files as we only presented the firs encountered error.

Even though our study has its shortcomings, limitations are to be expected in proposing a first-of-its-kind case study. We have proposed a well performing initial methodology and provided various insights that can readily be used in future research as to quickly improve upon our model (Section 5).

5 Future Research

Although our best performing model would not translate well into a user product, our study is a first-of-its-kind. Therefore, it leaves a lot of room for future research. One could argue that using an LLM might not be the best way to synthesize ORCA input files. Especially when keeping in mind that the loss of `GPT-3.5 Turbo` was not a good indicator of final classification performance. It would be interesting to have a look at whether other model types, like Code-BERT[73] in a classification settings, would outperform our LLM. Importantly, one would then lose the chat aspect of our approach, but this could still be achieved with a seperate LLM for that task resulting in a multimodal approach. Note that the ability to chat with our LLM about input files was not explored. It would be interesting to prompt the LLM in different ways to gauge its general ORCA knowledge. For instance, one could assess if the finetuned models perform better at explaining the contents of given input files or if they can suggest the proper keyword based on a single description line.

Additionally, based on our conclusion about the interaction between CoT and finetuning, it would be interesting to see how the other proposed prompt engineering techniques perform when used in combination with finetuning. It could be that one of the other implemented prompt engineering techniques would outperform CoT when combined with finetuning.

Moreover, an interesting research direction would be to try and make the generation of user prompts more generalisable for other DSLs, as our approach uses a hard-coded way of mapping the terms in an ORCA input file to their description. One could, for instance, propose an end-to-end trainable code captioning model[25,68] that is trained to map the code snippets of the DSL to user prompts. Another viable way of altering the user prompt generation would be to not include the keywords in the prompt. While perhaps unrealistic from a user point of view, it would allow for the model to learn mapping keyword-descriptions to keywords in that we prohibit it from directly copying keywords from the user prompt.

There is also the opportunity to extend upon our framework. Many more intermediate steps could be taken to improve the final input file that gets synthesized. For instance, it would be rather straightforward to implement the removal of duplicate keywords, or to exchange synthesised keywords that are not in the extracted documentation with ones that are similar to them (or even have the LLM choose from similar keywords in a multi-shot way). Similarly, it would be interesting to see how the model handles the feedback provided by ORCA with respect to the encountered errors. In that regard, it would be intriguing to see whether a feedback loop as described by Skreta *et al.*[82] would improve ORCA code synthesis.

Lastly we highlight the fact that active learning[83] could be used in our methodology, as it fits the DSL-context well. During finetuning, one could have the LLM decide what user prompts it wants to learn from so as to optimally make use of the limited amount of data. This is particularly alluring because our synthetic data generation process may have resulted in datasets containing a significant amount of

uninformative data.

Appendix

A Prompts

A.1 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
   Weigend's 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 Hartree-Fock 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.2 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.

[FORMAT DESCRIPTION AND EXAMPLES FROM THE 'BASIC' PROMPT]

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.3 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.

[FORMAT DESCRIPTION AND EXAMPLES FROM THE 'BASIC' PROMPT]

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.4 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.

[FORMAT DESCRIPTION AND EXAMPLES FROM THE 'BASIC' PROMPT]

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.
```

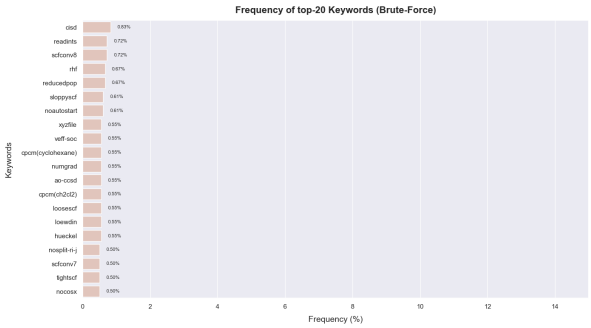
A.5 Tree of Thoughts

```
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.

[FORMAT DESCRIPTION AND EXAMPLES FROM THE 'BASIC' PROMPT]

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.
```

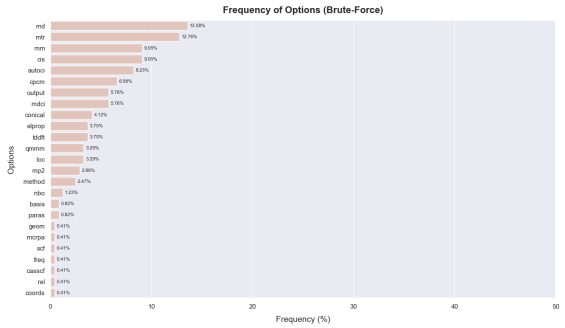
2. Evaluate each initial thought for relevance and accuracy. Select the most appropriate keywords and settings.
3. Consolidate the best options into a coherent input file. Review and refine to ensure accuracy and completeness.

Generate the input file by following these steps and ensure each step is explained.

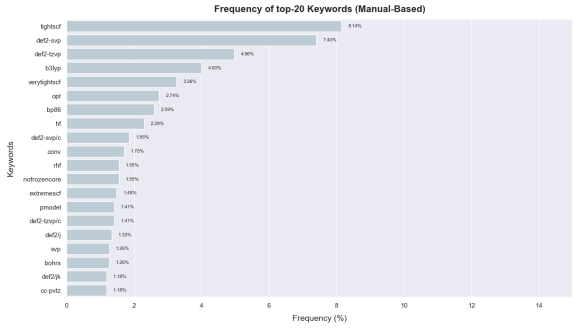
B Dataset Overviews



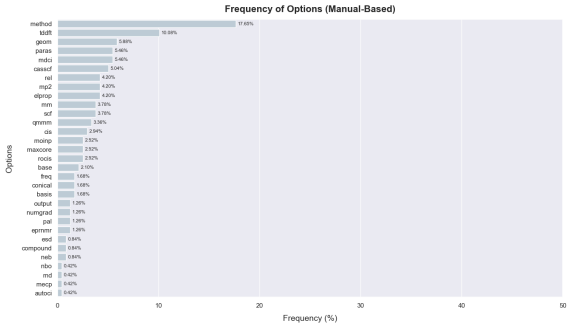
(a)



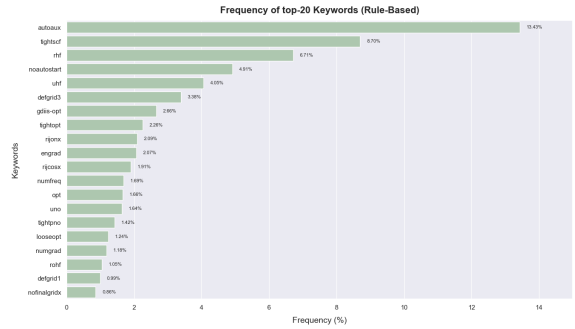
(b)



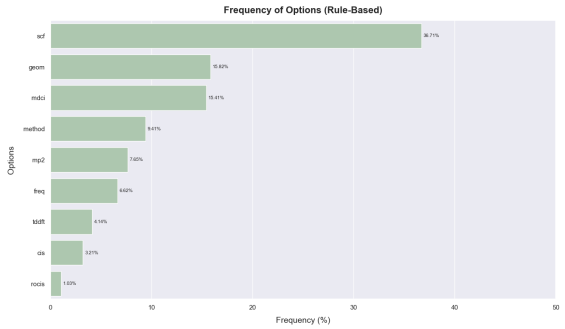
(c)



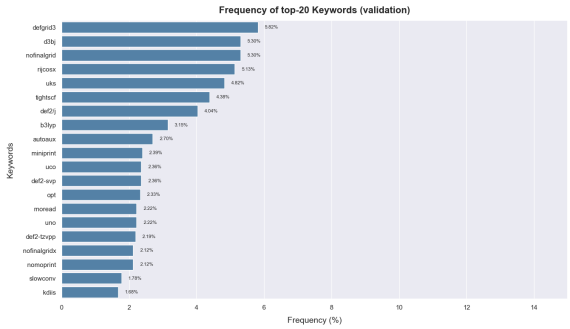
(d)



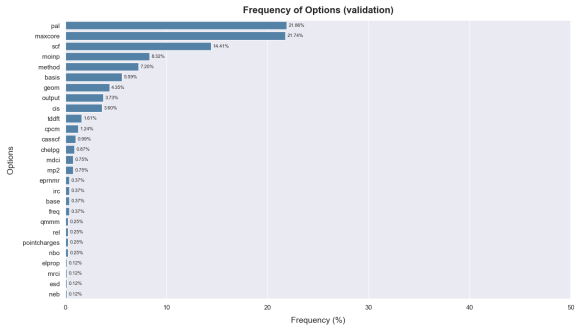
(e)



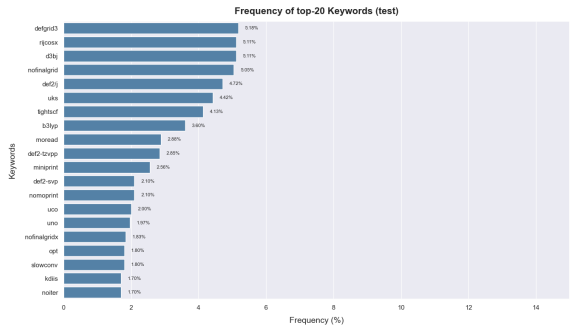
(f)



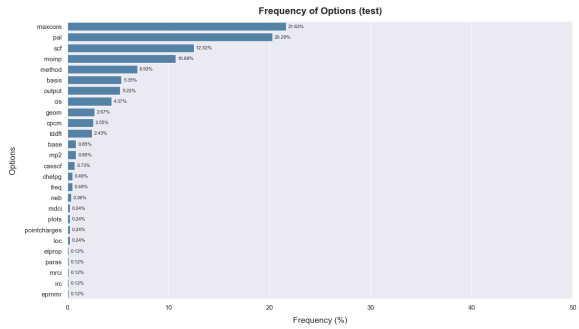
(g)



(h)



(i)



(j)

Figure 11: Overview of the top-20 most occurring keywords (a, c, e, g, i) and all frequencies for the different options (b, d, f, h, j) for the brute-force, manual-based and rule-based, validation and test datasets, respectively.

References

[1] S. Liu, W. Nie, C. Wang, J. Lu, Z. Qiao, L. Liu, J. Tang, C. Xiao and A. Anandkumar, *Nature Machine Intelligence*, 2023, **5**, 1447–1457.

[2] S. Wang, Y. Guo, Y. Wang, H. Sun and J. Huang, Proceedings of the 10th ACM international conference on bioinformatics, computational biology and health informatics, 2019, pp. 429–436.

[3] S. Chithrananda, G. Grand and B. Ramsundar, *arXiv*, 2020, 2010.09885.

[4] W. Ahmad, E. Simon, S. Chithrananda, G. Grand and B. Ramsundar, *arXiv*, 2022, 2209.01712.

[5] D. Weininger, *Journal of chemical information and computer sciences*, 1988, **28**, 31–36.

[6] G. M. Hocky and A. D. White, *Digital discovery*, 2022, **1**, 79–83.

[7] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever and W. Zaremba, *arXiv*, 2021, 2107.03374.

[8] A. D. White, G. M. Hocky, H. A. Gandhi, M. Ansari, S. Cox, G. P. Wellawatte, S. Sasmal, Z. Yang, K. Liu, Y. Singh and W. J. Peña Ccoa, *Digital Discovery*, 2023, **2**, 368–376.

[9] D. A. Boiko, R. MacKnight, B. Kline and G. Gomes, *Nature*, 2023, **624**, 570–578.

[10] A. M. Bran, S. Cox, O. Schilter, C. Baldassari, A. D. White and P. Schwaller, *Nature Machine Intelligence*, 2024, 1–11.

[11] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan and Y. Cao, *arXiv*, 2023, 2210.03629.

[12] J. Wei, X. Wang, D. Schuurmans, M. Bosma, b. ichter, F. Xia, E. Chi, Q. V. Le and D. Zhou, Advances in Neural Information Processing Systems, 2022, pp. 24824–24837.

[13] K. Valmeekam, M. Marquez, S. Sreedharan and S. Kambhampati, *Advances in Neural Information Processing Systems*, 2023, **36**, 75993–76005.

[14] D. E. Wilkins and K. L. Myers, *Journal of Logic and Computation*, 1995, **5**, 731–761.

[15] J. A. Hendler, A. Tate and M. Drummond, *AI magazine*, 1990, **11**, 61–61.

[16] A. Mumuni and F. Mumuni, *Array*, 2022, **16**, 100258.

[17] P. Eigenschink, T. Reutterer, S. Vamosi, R. Vamosi, C. Sun and K. Kalcher, *IEEE Access*, 2023, **11**, 47304–47320.

[18] C. Chokwitthaya, Y. Zhu, S. Mukhopadhyay and A. Jafari, Construction Research Congress 2020, 2020, pp. 1251–1260.

[19] R. P. Pargas, M. J. Harrold and R. R. Peck, *Software testing, verification and reliability*, 1999, **9**, 263–282.

[20] K. Houkjær, K. Torp and R. Wind, Proceedings of the 32nd international conference on Very large data bases, 2006, pp. 1243–1246.

[21] P. Yin and G. Neubig, Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Vancouver, Canada, 2017, pp. 440–450.

[22] E. Parisotto, A. rahman Mohamed, R. Singh, L. Li, D. Zhou and P. Kohli, *arXiv*, 2016, 1611.01855.

[23] R. Shin, N. Kant, K. Gupta, C. Bender, B. Trabucco, R. Singh and D. Song, *arXiv*, 2019, 1912.12345.

[24] U. Alon, S. Brody, O. Levy and E. Yahav, *arXiv*, 2019, 1808.01400.

[25] V. Karia, A. Mukherjee, Z. Doshi, V. Pendse and V. Chang, *Code2Cap: Automated Code Captioning*, 2019, `https://fabrice.harel-canada.com/archives/projects/code2cap/code2cap.pdf`.

[26] M. Allamanis, D. Tarlow, A. Gordon and Y. Wei, Proceedings of the 32nd International Conference on Machine Learning, Lille, France, 2015, pp. 2123–2132.

[27] A. Bauer, S. Trapp, M. Stenger, R. Leppich, S. Kounev, M. Leznik, K. Chard and I. Foster, *arXiv*, 2024, 2401.02524.

[28] D. OBrien, S. Biswas, S. M. Imtiaz, R. Abdalkareem, E. Shihab and H. Rajan, Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–13.

[29] N. Koul, *Prompt Engineering for Large Language Models*, `https://books.google.at/books?id=e9_jEAAAQBAJ`.

[30] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang and M. R. Lyu, Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering, 2022, pp. 382–394.

[31] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever and D. Amodei, Advances in Neural Information Processing Systems, 2020, pp. 1877–1901.

[32] J. Chen, L. Chen, H. Huang and T. Zhou, *arXiv*, 2023, 2304.03262.

[33] Z. Shao, Y. Gong, Y. Shen, M. Huang, N. Duan and W. Chen, International Conference on Machine Learning, 2023, pp. 30706–30775.

[34] Z. Yu, L. He, Z. Wu, X. Dai and J. Chen, *arXiv*, 2023, 2310.04959.

[35] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal and A. Chadha, *arXiv*, 2024, 2402.07927.

[36] C. Li, J. Liang, A. Zeng, X. Chen, K. Hausman, D. Sadigh, S. Levine, L. Fei-Fei, F. Xia and B. Ichter, *arXiv*, 2023, 2312.04474.

[37] J. Li, G. Li, Y. Li and Z. Jin, *arXiv*, 2023, 2305.06599.

[38] S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths, Y. Cao and K. Narasimhan, Advances in Neural Information Processing Systems, 2023, pp. 11809–11822.

[39] N. Dainese, M. Merler, M. Alakuijala and P. Marttinen, *arXiv*, 2024, 2405.15383.

[40] M. Besta, N. Blach, A. Kubicek, R. Gerstenberger, M. Podstawski, L. Gianinazzi, J. Gajda, T. Lehmann, H. Niewiadomski, P. Nyczyk and T. Hoefler, *Proceedings of the AAAI Conference on Artificial Intelligence*, 2024, **38**, 17682–17690.

[41] R. Bavishi, C. Lemieux, K. Sen and I. Stoica, *Proceedings of the ACM on Programming Languages*, 2021, **5**, 1–29.

[42] X. Gao, T. Kim, M. D. Wong, D. Raghunathan, A. K. Varma, P. G. Kannan, A. Sivaraman, S. Narayana and A. Gupta, Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, 2020, pp. 44–61.

[43] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, L. Zhang, Z. Li and Y. Ma, *arXiv*, 2024, 2404.00971.

[44] Y. Tian, W. Yan, Q. Yang, Q. Chen, W. Wang, Z. Luo and L. Ma, *arXiv*, 2024, 2405.00253.

[45] S. Dhuliawala, M. Komeili, J. Xu, R. Raileanu, X. Li, A. Celikyilmaz and J. Weston, Findings of the Association for Computational Linguistics ACL 2024, Bangkok, Thailand and virtual meeting, 2024, pp. 3563–3578.

[46] S. Kouemo Ngassom, A. Moradi Dakhel, F. Tambon and F. Khomh, Proceedings of the 1st ACM International Conference on AI-Powered Software, New York, NY, USA, 2024, p. 122–130.

[47] X. Liu, K. Ji, Y. Fu, W. Tam, Z. Du, Z. Yang and J. Tang, Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), Dublin, Ireland, 2022, pp. 61–68.

[48] T. Shin, Y. Razeghi, R. L. Logan IV, E. Wallace and S. Singh, Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), Online, 2020, pp. 4222–4235.

[49] T. Ridnik, D. Kredo and I. Friedman, *arXiv*, 2024, 2401.08500.

[50] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel and D. Kiela, Advances in Neural Information Processing Systems, 2020, pp. 9459–9474.

[51] V. Karpukhin, B. Oguz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen and W.-t. Yih, Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), Online, 2020, pp. 6769–6781.

[52] P. C. Verhoef, T. Broekhuizen, Y. Bart, A. Bhattacharya, J. Q. Dong, N. Fabian and M. Haenlein, *Journal of business research*, 2021, **122**, 889–901.

[53] K. Shuster, S. Poff, M. Chen, D. Kiela and J. Weston, Findings of the Association for Computational Linguistics: EMNLP 2021, Punta Cana, Dominican Republic, 2021, pp. 3784–3803.

[54] W. Wang, Y. Wang, S. Joty and S. C. Hoi, Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, New York, NY, USA, 2023, p. 146–158.

[55] Y. Wang, W. Wang, S. Joty and S. C. Hoi, Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, Online and Punta Cana, Dominican Republic, 2021, pp. 8696–8708.

[56] M. R. Parvez, W. U. Ahmad, S. Chakraborty, B. Ray and K. Chang, Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021, 2021, pp. 2719–2734.

[57] S. Lu, N. Duan, H. Han, D. Guo, S. Hwang and A. Svyatkovskiy, Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022, 2022, pp. 6227–6240.

[58] H. Tan, Q. Luo, L. Jiang, Z. Zhan, J. Li, H. Zhang and Y. Zhang, *arXiv*, 2024, 2405.07530.

[59] A. Slivkins, *Found. Trends Mach. Learn.*, 2019, **12**, 1–286.

[60] N. Baumann, J. S. Diaz, J. Michael, L. Netz, H. Nqiri, J. Reimer and B. Rumpe, Modellierung 2024 - Workshop Proceedings, Potsdam, Germany, March 12-15, 2024, 2024, p. 7.

[61] S. Zhou, U. Alon, F. F. Xu, Z. Wang, Z. Jiang and G. Neubig, *arXiv*, 2023, 2207.05987.

[62] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, M. Wang and H. Wang, *arXiv*, 2024, 2312.10997.

[63] K. Ogueji, Y. Zhu and J. Lin, Proceedings of the 1st Workshop on Multilingual Representation Learning, Punta Cana, Dominican Republic, 2021, pp. 116–126.

[64] J. Dodge, G. Ilharco, R. Schwartz, A. Farhadi, H. Hajishirzi and N. Smith, *arXiv*, 2020, 2002.06305.

[65] J. Huang, S. Gu, L. Hou, Y. Wu, X. Wang, H. Yu and J. Han, Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023, 2023, pp. 1051–1068.

[66] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le and C. Sutton, *arXiv*, 2021, 2108.07732.

[67] G. Dong, H. Yuan, K. Lu, C. Li, M. Xue, D. Liu, W. Wang, Z. Yuan, C. Zhou and J. Zhou, Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Bangkok, Thailand, 2024, pp. 177–198.

[68] S. Chaudhary, *Code Alpaca: An Instruction-following LLaMA model for code generation*, `https://github.com/sahil280114/codealpaca`, 2023.

[69] S. Zhang, L. Dong, X. Li, S. Zhang, X. Sun, S. Wang, J. Li, R. Hu, T. Zhang, F. Wu and G. Wang, *arXiv*, 2024, 2308.10792.

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

[71] Q. Sun, T. C. Berkelbach, N. S. Blunt, G. H. Booth, S. Guo, Z. Li, J. Liu, J. D. McClain, E. R. Sayfutyarova, S. Sharma, S. Wouters and G. K.-L. Chan, *WIREs Computational Molecular Science*, 2018, **8**, e1340.

[72] F. Neese, F. Wennmohs, U. Becker and C. Riplinger, *The Journal of Chemical Physics*, 2020, **152**, 224108.

[73] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang and M. Zhou, Findings of the Association for Computational Linguistics: EMNLP 2020, Online, 2020, pp. 1536–1547.

[74] J. Ye, X. Chen, N. Xu, C. Zu, Z. Shao, S. Liu, Y. Cui, Z. Zhou, C. Gong, Y. Shen, J. Zhou, S. Chen, T. Gui, Q. Zhang and X. Huang, *arXiv*, 2023, 2303.10420.

[75] W.-L. Chiang, L. Zheng, Y. Sheng, A. N. Angelopoulos, T. Li, D. Li, H. Zhang, B. Zhu, M. Jordan, J. E. Gonzalez and I. Stoica, *arXiv*, 2024, 2403.04132.

[76] M. Reid, N. Savinov, D. Teplyashin, Dmitry, Lepikhin, T. Lillicrap, J. baptiste Alayrac, R. Soricut, A. Lazaridou, O. Firat, J. Schrittwieser, I. Antonoglou, R. Anil, S. Borgeaud, A. Dai, K. Millican, E. Dyer, M. Glaese, T. Sottiaux, B. Lee, F. Viola, M. Reynolds, Y. Xu, J. Molloy, J. Chen, M. Isard, P. Barham, T. Hennigan, R. McIlroy, M. Johnson, J. Schalkwyk, E. Collins, E. Rutherford, E. Moreira, K. Ayoub, M. Goel, C. Meyer, G. Thornton, Z. Yang, H. Michalewski, Z. Abbas, N. Schucher, A. Anand, R. Ives, J. Keeling, K. Lenc, S. Haykal, S. Shakeri, P. Shyam, A. Chowdhery, R. Ring, S. Spencer, E. Sezener, L. Vilnis, O. Chang, N. Morioka, G. Tucker, C. Zheng, O. Woodman, N. Attaluri, T. Kocisky, E. Eltyshev, X. Chen, T. Chung, V. Selo, S. Brahma, P. Georgiev, A. Slone, Z. Zhu, J. Lottes, S. Qiao, B. Caine, S. Riedel, A. Tomala, M. Chadwick, J. Love, P. Choy, S. Mittal, N. Houlsby, Y. Tang, M. Lamm, L. Bai, Q. Zhang, L. He, Y. Cheng, P. Humphreys, Y. Li, S. Brin, A. Cassirer, Y. Miao, L. Zilka, T. Tobin, K. Xu, L. Proleev, D. Sohn, A. Magni, L. A. Hendricks, I. Gao, S. Ontanon, O. Bunyan, N. Byrd, A. Sharma, B. Zhang, M. Pinto, R. Sinha, H. Mehta, D. Jia, S. Caelles, A. Webson, A. Morris, B. Roelofs, Y. Ding, R. Strudel, X. Xiong, M. Ritter, M. Dehghani, R. Chaabouni, A. Karmarkar, G. Lai, F. Mentzer, B. Xu, Y. Li, Y. Zhang, T. L. Paine, A. Goldin, B. Neyshabur, K. Baumli, A. Levskaya, M. Laskin, W. Jia, J. W. Rae, K. Xiao, A. He, S. Giordano, L. Yagati, J.-B. Lespiau, P. Natsev, S. Ganapathy, F. Liu, D. Martins, N. Chen, Y. Xu, M. Barnes, R. May, A. Vezer, J. Oh, K. Franko, S. Bridgers, R. Zhao, B. Wu, B. Mustafa, S. Sechrist, E. Parisotto, T. S. Pillai, C. Larkin, C. Gu, C. Sorokin, M. Krikun, A. Guseynov, J. Landon, R. Datta, A. Pritzel, P. Thacker, F. Yang, K. Hui, A. Hauth, C.-K. Yeh, D. Barker, J. Mao-Jones, S. Austin, H. Sheahan, P. Schuh, J. Svensson, R. Jain, V. Ramasesh, A. Briukhov, D.-W. Chung, T. von Glehn, C. Butterfield, P. Jhakra, M. Wiethoff, J. Frye, J. Grimstad, B. Changpinyo, C. L. Lan, A. Bortsova, Y. Wu, P. Voigtlaender, T. Sainath, S. Gu, C. Smith, W. Hawkins, K. Cao, J. Besley, S. Srinivasan, M. Omernick, C. Gaffney, G. Surita, R. Burnell, B. Damoc, J. Ahn, A. Brock, M. Pajarskas, A. Petrushkina, S. Noury, L. Blanco, K. Swersky, A. Ahuja, T. Avrahami, V. Misra, R. de Liedekerke, M. Iinuma, A. Polozov, S. York, G. van den Driessche, P. Michel, J. Chiu, R. Blevins, Z. Gleicher, A. Recasens, A. Rrustemi, E. Gribovskaya, A. Roy, W. Gworek, S. M. R. Arnold, L. Lee, J. Lee-Thorp, M. Maggioni, E. Piqueras, K. Badola, S. Vikram, L. Gonzalez, A. Baddepudi, E. Senter, J. Devlin, J. Qin, M. Azzam, M. Trebacz, M. Polacek, K. Krishnakumar, S. yiin Chang, M. Tung, I. Penchev, R. Joshi, K. Olszewska, C. Muir, M. Wirth, A. J. Hartman, J. Newlan, S. Kashem, V. Bolina, E. Dabir, J. van Amersfoort, Z. Ahmed, J. Cobon-Kerr, A. Kamath, A. M. Hrafnkelsson, L. Hou, I. Mackinnon, A. Frechette, E. Noland, X. Si, E. Taropa, D. Li, P. Crone, A. Gulati, S. Cevey, J. Adler, A. Ma, D. Silver, S. Tokumine, R. Powell, S. Lee, K. Vodrahalli, S. Hassan, D. Mincu, A. Yang, N. Levine, J. Brennan, M. Wang, S. Hodkinson, J. Zhao, J. Lipschultz, A. Pope, M. B. Chang, C. Li, L. E. Shafey, M. Paganini, S. Douglas, B. Bohnet, F. Pardo, S. Odoom, M. Rosca, C. N. dos Santos, K. Soparkar, A. Guez, T. Hudson, S. Hansen, C. Asawaroengchai, R. Addanki, T. Yu, W. Stokowiec, M. Khan, J. Gilmer, J. Lee, C. G. Bostock, K. Rong, J. Caton, P. Pejman, F. Pavetic, G. Brown, V. Sharma, M. Lučić, R. Samuel, J. Djolonga, A. Mandhane, L. L. Sjösund, E. Buchatskaya, E. White, N. Clay, J. Jiang, H. Lim, R. Hemsley, Z. Cankara, J. Labanowski, N. D. Cao, D. Steiner, S. H. Hashemi, J. Austin, A. Gergely, T. Blyth, J. Stanton, K. Shivakumar, A. Siddhant, A. Andreassen, C. Araya, N. Sethi, R. Shivanna, S. Hand, A. Bapna, A. Khodaei, A. Miech, G. Tanzer, A. Swing, S. Thakoor, L. Aroyo, Z. Pan, Z. Nado, J. Sygnowski, S. Winkler, D. Yu, M. Saleh, L. Maggiore, Y. Bansal, X. Garcia, M. Kazemi, P. Patil, I. Dasgupta, I. Barr, M. Giang, T. Kagohara, I. Danihelka, A. Marathe, V. Feinberg, M. Elhawaty, N. Ghelani,

D. Horgan, H. Miller, L. Walker, R. Tanburn, M. Tariq, D. Shrivastava, F. Xia, Q. Wang, C.-C. Chiu, Z. Ashwood, K. Baatarsukh, S. Samangooei, R. L. Kaufman, F. Alcober, A. Stjerngren, P. Komarek, K. Tsihlas, A. Boral, R. Comanescu, J. Chen, R. Liu, C. Welty, D. Bloxwich, C. Chen, Y. Sun, F. Feng, M. Mauger, X. Dotiwalla, V. Hellendoorn, M. Sharman, I. Zheng, K. Haridasan, G. Barth-Maron, C. Swanson, D. Rogozińska, A. Andreev, P. K. Rubenstein, R. Sang, D. Hurt, G. Elsayed, R. Wang, D. Lacey, A. Ilić, Y. Zhao, A. Iwanicki, A. Lince, A. Chen, C. Lyu, C. Lebsack, J. Griffith, M. Gaba, P. Sandhu, P. Chen, A. Koop, R. Rajwar, S. H. Yeganeh, S. Chang, R. Zhu, S. Radpour, E. Davoodi, V. I. Lei, Y. Xu, D. Toyama, C. Segal, M. Wicke, H. Lin, A. Bulanova, A. P. Badia, N. Rakićević, P. Sprechmann, A. Filos, S. Hou, V. Campos, N. Kassner, D. Sachan, M. Fortunato, C. Iwuanyanwu, V. Nikolaev, B. Lakshminarayanan, S. Jazayeri, M. Varadarajan, C. Tekur, D. Fritz, M. Khalman, D. Reitter, K. Dasgupta, S. Sarcar, T. Ornduff, J. Snaider, F. Huot, J. Jia, R. Kemp, N. Trdin, A. Vijayakumar, L. Kim, C. Angermueller, L. Lao, T. Liu, H. Zhang, D. Engel, S. Greene, A. White, J. Austin, L. Taylor, S. Ashraf, D. Liu, M. Georgaki, I. Cai, Y. Kulizhskaya, S. Goenka, B. Saeta, Y. Xu, C. Frank, D. de Cesare, B. Robenek, H. Richardson, M. Alnahlawi, C. Yew, P. Ponnapalli, M. Tagliasacchi, A. Korchemniy, Y. Kim, D. Li, B. Rosgen, K. Levin, J. Wiesner, P. Banzal, P. Srinivasan, H. Yu, Çağlar Ünlü, D. Reid, Z. Tung, D. Finchelstein, R. Kumar, A. Elisseeff, J. Huang, M. Zhang, R. Aguilar, M. Giménez, J. Xia, O. Dousse, W. Gierke, D. Yates, K. Jalan, L. Li, E. Latorre-Chimoto, D. D. Nguyen, K. Durden, P. Kallakuri, Y. Liu, M. Johnson, T. Tsai, A. Talbert, J. Liu, A. Neitz, C. Elkind, M. Selvi, M. Jasarevic, L. B. Soares, A. Cui, P. Wang, A. W. Wang, X. Ye, K. Kallarackal, L. Loher, H. Lam, J. Broder, D. Holtmann-Rice, N. Martin, B. Ramadhana, M. Shukla, S. Basu, A. Mohan, N. Fernando, N. Fiedel, K. Paterson, H. Li, A. Garg, J. Park, D. Choi, D. Wu, S. Singh, Z. Zhang, A. Globerson, L. Yu, J. Carpenter, F. de Chaumont Quitry, C. Radebaugh, C.-C. Lin, A. Tudor, P. Shroff, D. Garmon, D. Du, N. Vats, H. Lu, S. Iqbal, A. Yakubovich, N. Tripuraneni, J. Manyika, H. Qureshi, N. Hua, C. Ngani, M. A. Raad, H. Forbes, J. Stanway, M. Sundararajan, V. Ungureanu, C. Bishop, Y. Li, B. Venkatraman, B. Li, C. Thornton, S. Scellato, N. Gupta, Y. Wang, I. Tenney, X. Wu, A. Shenoy, G. Carvajal, D. G. Wright, B. Bariach, Z. Xiao, P. Hawkins, S. Dalmia, C. Farabet, P. Valenzuela, Q. Yuan, A. Agarwal, M. Chen, W. Kim, B. Hulse, N. Dukkipati, A. Paszke, A. Bolt, K. Choo, J. Beattie, J. Prendki, H. Vashisht, R. Santamaria-Fernandez, L. C. Cobo, J. Wilkiewicz, D. Madras, A. Elqursh, G. Uy, K. Ramirez, M. Harvey, T. Liechty, H. Zen, J. Seibert, C. H. Hu, A. Khorlin, M. Le, A. Aharoni, M. Li, L. Wang, S. Kumar, N. Casagrande, J. Hoover, D. E. Badawy, D. Soergel, D. Vnukov, M. Miecnikowski, J. Simsa, P. Kumar, T. Sellam, D. Vlasic, S. Daruki, N. Shabat, J. Zhang, G. Su, J. Zhang, J. Liu, Y. Sun, E. Palmer, A. Ghaffarkhah, X. Xiong, V. Cotruta, M. Fink, L. Dixon, A. Sreevatsa, A. Goedeckemeyer, A. Dimitriev, M. Jafari, R. Crocker, N. FitzGerald, A. Kumar, S. Ghemawat, I. Philips, F. Liu, Y. Liang, R. Sterneck, A. Repina, M. Wu, L. Knight, M. Georgiev, H. Lee, H. Askham, A. Chakladar, A. Louis, C. Crous, H. Cate, D. Petrova, M. Quinn, D. Owusu-Afriyie, A. Singhal, N. Wei, S. Kim, D. Vincent, M. Nasr, C. A. Choquette-Choo, R. Tojo, S. Lu, D. de Las Casas, Y. Cheng, T. Bolukbasi, K. Lee, S. Fatehi, R. Ananthanarayanan, M. Patel, C. Kaed, J. Li, S. R. Belle, Z. Chen, J. Konzelmann, S. Põder, R. Garg, V. Koverkathu, A. Brown, C. Dyer, R. Liu, A. Nova, J. Xu, A. Walton, A. Parrish, M. Epstein, S. McCarthy, S. Petrov, D. Hassabis, K. Kavukcuoglu, J. Dean and O. Vinyals, *arXiv*, 2024, 2403.05530.

[77] M. Enis and M. Hopkins, *arXiv*, 2024, 2404.13813.

[78] DeepSeek-AI, A. Liu, B. Feng, B. Wang, B. Wang, B. Liu, C. Zhao, C. Dengr, C. Ruan, D. Dai, D. Guo, D. Yang, D. Chen, D. Ji, E. Li, F. Lin, F. Luo, G. Hao, G. Chen, G. Li, H. Zhang, H. Xu, H. Yang, H. Zhang, H. Ding, H. Xin, H. Gao, H. Li, H. Qu, J. L. Cai, J. Liang, J. Guo, J. Ni, J. Li, J. Chen, J. Yuan, J. Qiu, J. Song, K. Dong, K. Gao, K. Guan, L. Wang, L. Zhang, L. Xu, L. Xia, L. Zhao, L. Zhang, M. Li, M. Wang, M. Zhang, M. Zhang, M. Tang, M. Li, N. Tian, P. Huang, P. Wang, P. Zhang, Q. Zhu, Q. Chen, Q. Du, R. J. Chen, R. L. Jin, R. Ge, R. Pan, R. Xu, R. Chen, S. S. Li, S. Lu, S. Zhou, S. Chen, S. Wu, S. Ye, S. Ma, S. Wang, S. Zhou, S. Yu, S. Zhou, S. Zheng, T. Wang, T. Pei, T. Yuan, T. Sun, W. L. Xiao, W. Zeng, W. An, W. Liu, W. Liang, W. Gao, W. Zhang, X. Q. Li, X. 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, Y. Wu, Y. K. Li, Y. X. Wei, Y. X. Zhu, Y. 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, Z. Z. Ren, 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 and Z. Xie, *arXiv*, 2024, 2405.04434.

[79] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave and G. Lample, *arXiv*, 2023, 2302.13971.

[80] M. Álvarez-Moreno, C. de Graaf, N. Lopez, F. Maseras, J. M. Poblet and C. Bo, *Journal of chemical information and modeling*, 2015, **55**, 95–103.

[81] S. Konstantinidis, *Information and Computation*, 2007, **205**, 1307–1316.

[82] M. Skreta, N. Yoshikawa, S. Arellano-Rubach, Z. Ji, L. B. Kristensen, K. Darvish, A. Aspuru-Guzik, F. Shkurti and A. Garg, *arXiv*, 2023, 2303.14100.

[83] C. Schröder and A. Niekler, *arXiv*, 2020, 2008.07267.