Heidelberg University

Institute of Computer Science

Artificial Intelligence for Programming (AIP)

Master's Thesis

Interrupt-Guided Neural Code Completion with Static Analysis of Dependencies

Name:	Niklas Henrik Loeser
Matrikelnummer:	3737339
Betreuer:	Prof. Dr. Artur Andrzejak
Datum der Abgabe:	2024-09-30

I hereby certify that I have written the work myself and that I have not used any sources or aids other than those specified, and that I have marked what has been taken over from other people's works, either verbatim or in terms of content, as foreign. I also certify that the electronic version of my thesis transmitted completely corresponds in content and wording to the printed version. I agree that this electronic version is being checked for plagiarism at the university using plagiarism software.

Heidelberg, the 2024-09-27

_____

Niklas Henrik Loeser

Abstract

- English -

Large Language Models (LLMs) have proven themselves successful in many fields, including code generation. Models complement regular Integrated Development Environment (IDE)-based code completion by predicting a completion based on the surrounding code context. Even so, LLMs perform best on data represented in their training data. Training is usually done once, as continuous training is slow and expensive. Models thus struggle with ever-changing dependencies, as they provide code completions based on outdated dependency versions. Moreover, even when predicting code of an older version represented in training data, models have issues distinguishing different versions of a dependency. This thesis retrieves accurate code context of dependencies using IDE tooling, injecting the context during model inference. The injected context guides the model to generate accurate completions based on the exact project environment.

We first propose a novel method, Interrupt-Based Meta-Strategy (IBMS), to efficiently inject context into the prompt during the generation. Natural language context such as code documentation is best fit for injection, as more structured context may be used to more directly guide the model. This method is universal and may be used with any context formattable into the prompt. It is an improvement over multi-step generation meta-strategies, as the performance overhead is greatly reduced, as the model only generates code once.

Next, the thesis shows that the model may often ignore context injected into the prompt, such as deprecation information, due to inherent model bias. To remediate the issue, the thesis presents a novel approach, combining the IBMS with logits manipulation and IDE tooling. IDE tooling is used to retrieve deprecation information, code completion entries and function signatures during the generation of the completion, watching the model in realtime. When appropriate, the retrieved context is injected into the prompt using the IBMS, softly guiding the model to generate completions based on the exact project environment. To reduce the effect of the model bias towards deprecated items in the training data, the predicted token id scores are reranked based on the retrieved context, such as deprecation information and completion items.

Lastly, the thesis presents DependencyEval, a handcrafted dataset and evaluation framework containing Python code completion tasks. The dataset entries contain information describing a reproducable code environment, including the Python version and exact dependency versions. DependencyEval is used to evaluate the approach in different configurations, comparing the results against the baseline.

Abstract

- German -

Large Language Models (LLMs) haben sich in vielen Bereichen bewährt, z.B. bei der Codegenerierung. Modelle ergänzen die reguläre Integrated Development Environment (IDE)-basierte Codevervollständigung, indem sie eine Vervollständigung basierend auf dem umgebenden Codekontext vorhersagen. Dennoch sind LLMs am besten für Daten geeignet, die in ihren Trainingsdaten enthalten sind. Das Training wird in der Regel einmal durchgeführt, da kontinuierliches Training langsam und teuer ist. Modelle haben daher mit sich ständig ändernden Abhängigkeiten Probleme, da sie Codevervollständigungen auf der Grundlage veralteter Abhängigkeitsversionen liefern. Darüber hinaus haben die Modelle Probleme, selbst wenn verwendete Versionen von Abhängigkeiten in den Trainingsdaten sind. In dieser Arbeit wird daher der genaue Code-Kontext von Abhängigkeiten mithilfe von IDE-Tools ermittelt, und während Modellinferenz hinzugeführt. Dadurch wird das Modell angeleitet, genaue Vervollständigungen auf der Grundlage der genauen Projektumgebung zu generieren.

Wir schlagen zunächst die neue Methode Interrupt-Based Meta-Strategy (IBMS) vor, um den Kontext während der Generierung effizient in die Prompt zu integrieren. Natürlichsprachlicher Kontext, wie z.B. Code-Dokumentation, eignet sich am besten für die Injektion, da strukturierterer Kontext verwendet werden kann, um das Modell direkter zu steuern. Die Methode ist universell und kann mit jedem Kontext verwendet werden, der in die Prompt eingebaut werden kann. Sie stellt eine Verbesserung gegenüber Metastrategien mit mehreren Generierungsschritten dar, da die Laufzeit erheblich reduziert wird.

Als Nächstes stellt die Arbeit fest, dass das Modell oft den in die Eingabeaufforderung eingefügten Kontext ignoriert, wie z.B. Deprecationsinformationen, da das Modell eine ältere, während des Trainings erworbene Version der Abhängigkeit bevorzugt. Um dieses Problem zu beheben, wird in dieser Arbeit ein neuer Ansatz vorgestellt, der IBMS mit der Manipulation von Logits und IDE-Tools kombiniert. Das IDE-Tooling wird eingesetzt, um während der Generierung der Vervollständigung Informationen über Deprecations, Codevervollständigungseinträge und Funktionssignaturen abzurufen und das Modell in Echtzeit zu beobachten. Gegebenenfalls wird der abgerufene Kontext mit Hilfe von IBMS in die Eingabeaufforderung eingefügt, wodurch das Modell sanft dazu angeleitet wird, Vervollständigungen auf der Grundlage der genauen Projektumgebung zu erzeugen. Um die Auswirkung des Modells auf veraltete Elemente in den Trainingsdaten zu reduzieren, werden die vorhergesagten Token-ID-Scores auf der Grundlage des abgerufenen Kontexts, wie z.B. Informationen über veraltete Elemente und Vervollständigungselemente, neu eingestuft.

Schließlich wird in dieser Arbeit DependencyEval vorgestellt, ein handgefertigter Datensatz und ein Evaluationsframework, der Aufgaben zur Vervollständigung von Python-Code enthält. Die Einträge des Datensatzes enthalten Informationen, die eine reproduzierbare Codeumgebung beschreiben, einschließlich der Python-Version und der genauen Versionen der Abhängigkeiten. DependencyEval wird verwendet, um den Ansatz in verschiedenen Konfigurationen zu evaluieren und die Ergebnisse mit der Baseline zu vergleichen.

Contents

1 Introduction 1
1.1 Motivation 1
1.2 Goals 2
1.3 Contributions 2

2 Background & Related Work 4
2.1 Causal Language Models 4
2.2 Text Generation Strategies 6
2.3 Controllable Generation 7
2.4 Transformers Library 11
2.5 Language Server Protocol 14
2.6 Related Work 19

3 Approach 24
3.1 Requirements 24
3.2 Analysis 26
3.3 Interrupt-Based Meta-Strategy 27
3.4 Language Server Protocol aided generation 32
3.5 Dataset 36

4 Implementation 43
4.1 Language Server Protocol aided generation 43
4.2 Dataset Implementation 64

5 Experimental Evaluation 75
5.1 Experiment Environment 75
5.2 Research Questions 77

6 Results 80

7 Conclusion 97
7.1 Recap 97
7.2 Outlook 98

Acronyms 100

Bibliography 101

Listings 105

Appendix 111
A Language Server Protocol 111
B DependencyEval Dataset 113
C Evaluation Result Tables 125
D Evaluation Result Details 127

1 Introduction

Code completion provides code suggestions to developers extending their incomplete code. The suggestions use multimodal data sources to combine a variety of information.

It is one of the most used productivity tools in IDEs. Every major IDE from Visual Studio Code, Jetbrains IDEs, Eclipse, Neovim and Emacs with plugins supports code completion. It reduces the mental burden of remembering method names, properties, their types and associated information, such as documentation and if the item is deprecated. In addition, it saves time by reducing written boilerplate code by suggesting code snippets. Using the code completion saves time spent searching for corresponding documentation outside the IDEs.

As Amann et al. [1] found, code completion is the most frequently used tool inside IDEs with a frequency of 19.7%. Moreover, developers spend 22.4% of their time inside IDEs navigating code, in contrast to 28.5% editing and executing the code. They hypothesize that a significant amount of time navigating code is spent due to a lack of understanding of the code. Moreover, 39.8% of the total time is spent outside the IDE, which likely includes time spent searching for documentation.

Good code completion is a crucial tool for developers, reducing time spent in total and reducing the time spent outside the IDE, further increasing the focus of the developer.

1.1 Motivation

Neural code completion tools, such as GitHub Copilot, have seen a massive adoption in the software development community.

Such tools promise to complement typical code completion tools by providing context-aware code completions spanning multiple lines. These multi-line completions reduce the need to write boilerplate code further, thus improving the productivity.

While typical code completion tools use static analysis to provide completions, neural code completion tools use Causal Language Models (CLMs) trained on code to predict the next tokens. As Ziegler et al. [2] have shown, the completions provided by Copilot serve as a template to tinker with.

Even if the completions provide a starting point, this work argues that current neural code completions do not use enough context to be accurate and are insufficient if used without the correct documentation.

CLMs are released together with a "cutoff date", which is the date until which the training data has been collected. These models are incapable of predicting code completions which rely on information released after the cutoff date. Even worse, the training data may contain information on a mixture of several old versions of the same item.

Widely used code packages are updated frequently, in which code is added, removed, or changed. Often the code is not removed directly, but marked as deprecated. It is still available, but should not be used anymore.

It is no surprise, that given a coding environment with specific versions of packages, the completions are often incapable of using correct code items. When using cutting-edge packages, the suggestions may not know anything, contain now renamed function parameters or use long deprecated methods. For legacy projects the reverse is true, as the neural code completion may suggest items from newer versions, which are not available in the project. As such, developers are forced to use the suggestions as a rough template, which they have to adjust to the correct documentation. Hence, developers use classical IDE tooling or consult documentation directly to check single suggested items.

Methods such as retraining the model with correct package information ad-hoc are not feasible, as they are slow and require a lot of computational resources. Other dynamic methods providing the model with up-to-date information are not flexible enough and do not provide enough context to be accurate yet. Additionally, some current methods are too slow to be used in a real-time coding environment.

1.2 Goals

This thesis aims to combine up-to-date information through static analysis with the prediction capabilities of CLMs. The combination shall provide the neural code completion with the contextual information needed, to suggest code fitting the local code environment. The contextual information shall be provided at specific instances to guide the model **away** from incorrect suggestions **towards** correct suggestions. Additionally, the performance impact of the proposed approach shall be kept low, to ensure a responsive system. This will enable integration into IDEs and other tools.

Initially, the thesis will introduce important concepts and technologies, from Causal Language Modeling to Controllable Generation, followed by important details on libraries implementing these concepts for use in the approach implementation. Moreover, related work will be introduced to provide a basis for the proposed approach.

Afterward, the thesis will introduce the requirements for a proposed approach, then analyze shortcomings of current approaches and propose a new approach to address these shortcomings. Possible solutions will be discussed and evaluated, to find a promising approach.

Then, the thesis will describe the components of the proposed approach.

Finally, the approach will be implemented and evaluated, according to the defined requirements.

At last, the thesis will conclude with a summary of the results and an outlook on future work.

1.3 Contributions

The main contributions of the thesis are:

1. Performant, robust and generalizable neural code completion using up-to-date package information (engineering contribution).
2. A novel Interrupt-Based Meta-Strategy (IBMS) for injecting context into the model (conceptual contribution).

3. DependencyEval, a dataset to evaluate functional and approach correctness (equally conceptual and engineering contribution).

The main contribution integrates information from the currently used packages and local repository into the code completion. The information is retrieved through the code completion and signature help functionality, provided by typical language servers, which generalize this work over different programming languages. The information includes deprecation information, preventing the model from completing old methods and guiding it towards replacements, using the provided documentation. Performance-wise, the framework only incurs a small performance penalty over the usual predicted code completion. The contribution combines the novel concept of an IBMS with the concepts of a language server and logits manipulation with code completion. Even so, the contribution is heavily engineering focused, as implementing the IBMS for code completion has proven to be a complex task, as it requires a high level of integration into the code completion. Moreover, many improvements were made to the implementation, improving the robustness and performance under different circumstances.

The second contribution is a novel IBMS, which may be used outside of code completion. It enables monitoring the prediction similar to Agrawal et al. [3], but then injects retrieved information at critical time points into the context "during" the generation. It is used to inject the retrieved information into the model, nudging it towards the correct completion. The IBMS is the main conceptual contribution of the thesis. It is novel, applies even outside of code completion and can be integrated into many different approaches to e.g. bring tooling feedback closer towards the model.

Lastly, the dataset evaluates if the completed code is functionally correct and uses the correct approach. The evaluation checks, if e.g. a deprecated method was used and then fails the completed code. The dataset includes hand-picked code pieces of rapidly changing packages, legacy packages or new and unused packages. Each item includes the Python version and pinned dependencies, to create a reproducible code environment. To secure the evaluation, the evaluation pipeline uses Docker containers. In addition, the evaluation generates a report and draws diagrams. It is equally a conceptual and engineering contribution, as the dataset is closely built around the concept of reproducible environments unseen in other datasets. The reproducibility increases the effort to implement an evaluation loop, forming the engineering contribution.

2 Background & Related Work

This chapter provides important background information and related work necessary to understand the main contributions of this thesis.

Firstly, CLMs are introduced in Section 2.1. Thereafter, Section 2.2 discusses commonly used text generation strategies. The Section 2.3 showcases strategies to modify the text generation of a fixed pretrained CLM. Section 2.4 displays important components of the popular Hugging Face library, which enables CLM workflows using all the techniques introduced in the sections before. Thereafter, Section 2.5 introduces the functionality of the Language Server Protocol (LSP). At last, Section 2.6 introduces previous work.

2.1 Causal Language Models

This section briefly introduces the main components of causal language modeling. It touches on topics such as the general modeling, autoregressivity, the problem of vocabulary and special tokens.

A language model determines the probability of a sequence of words. The following formula calculates the probability of the sequence of words $w_1 ... w_n$, where w_i is the i-th word [4]:

$$P(w_1 ... w_n). \tag{1}$$

The probability of a sequence of words can be broken down into the product of the probabilities of each word given the previous words [4]:

$$P(w_1 ... w_n) = \prod_{i=1}^{n} P(w_i | w_1 ... w_{i-1}). \tag{2}$$

Example 1: Language model predicting the last word of the famous English pangram.

Previous words	*"The quick brown fox jumps over the lazy _"*
Predicted next word	*"dog"*

CLMs are neural statistical language models. Statistical language models model the probability function as a parameterized function, which is learned from a given corpus. The parameterized function requires a high number of parameters, as many different word combinations are possible, many of them unseen in the corpus. To battle the high dimensionality of the problem, neural statistical language models are used, as they generalize well to unseen data [5]. Neural statistical language models are composed of the following components:

1. A distributed representation of words, called word feature vectors or embeddings. Given a fixed vocabulary V and m features, the embeddings are obtained from the embedding function:

$$f_{\text{embedding}} : V \to \mathbb{R}^m. \tag{3}$$

2. Parameterized probability function, working on the embeddings [6]:

$$P : \mathbb{R}^m \times ... \times \mathbb{R}^m \to \mathbb{R}^{|V|}. \tag{4}$$

The composition of both functions results in the model function:

$$f_{\text{model}}(w_1 ... w_n) = P\big(f_{\text{embedding}}(w_1) ... f_{\text{embedding}}(w_n)\big). \tag{5}$$

The model learns both functions from a given corpus. By learning the distributed representation of words using m features, the model can generalize well to unseen data while keeping the number of parameters low. The probability function represents a matrix of size $|V| \times m$, the feature matrix.

The next step in causal language modeling is the property autoregressivity. Decomposing the probability function into a series of next word predictions enables an open-end-generation. The model can predict the next word, which is appended to the input sequence afterward. This process is repeated for the new sequence, which serves as the new input sequence for the next word prediction.

An important aspect of the next word prediction is the strategy used to determine the next word from the word prediction. The next best token may not be the best choice over the whole future sequence. The chosen text generation strategy has a significant impact on the quality of the generated text. Several strategies will be introduced in the next section.

A problem in language modeling is the vocabulary size. While the vocabulary V of a CLM is fixed, the number of possible words in a language is infinite. Languages change frequently, with old words falling out of use and new words emerging. Prominent examples include teenage slang, technical terms, and word combinations [7]. Using a fixed vocabulary comprised of words makes it impossible to model the language accurately. In contrast, using a vocabulary at the character level is not feasible, as many eastern languages feature thousands of characters. Additionally, encoding a sentence of a western language into characters would require a sequence of many vocabulary "words". The solution lies in-between, using sub-word fragments as vocabulary items. Popular choices used in modern language models are Byte Pair Encoding (BPE) and SentencePiece [8, 9]. The vocabulary entries are called tokens, with the process of encoding natural language into tokens called tokenization. The reverse process is called detokenization. The tokenization function is defined as follows:

$$f_{\text{tokenization}}(\text{text}) = w_1 ... w_n, \tag{6}$$

where $w_i \in V$ are tokens in the vocabulary V and text is any natural text. Ideally, the tokenization is reversible:

$$\text{text} = f_{\text{detokenization}}(f_{\text{tokenization}}(\text{text})). \tag{7}$$

Example 2: Natural language encoded into tokens and token ids using a word-based vocabulary.

Vocabulary	*{"The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog", "_", ".", "<BOS>", "<EOS>"}*
Text	*"The quick brown fox jumps"*
Tokens ids	*{11, 0, 9, 1, 9, 2, 9, 3, 4}*
Tokens	*{"<BOS>", "The", "_", "quick", "_", "brown", "_", "fox", "_", "jumps"}*

Example 2 showcases the tokenization for the start of the famous English pangram. The token values correspond to the index in the given vocabulary. Not only does the vocabulary contain tokens for natural language, but also special tokens. These special tokens encode further semantic information for the CLM. Common special tokens are the Begin of Sequence (BOS), End of Sequence (EOS) and Padding (PAD) tokens:

BOS Marks the beginning of a natural text. Used to delimit several unrelated texts.

EOS Marks the end of a natural text. Used in combination with BOS. The text generation is stopped, when the model predicts the EOS token as the next token.

PAD Used to pad sequences from either side to a fixed length. This is necessary for the model to process several sequences of varying lengths at once.

Especially vital to modern language models is the decomposition of the probability function into a series of next word predictions. This enables easy unsupervised training through the following learning objective: maximize the likelihood of the next word given the previous words over the corpus. To increase the stability of the training, the log-likelihood is maximized instead of the likelihood [10, 11]:

$$\text{argmax}_\theta \sum_{i=1}^{n} \log P_\theta(w_i | w_1 ... w_{i-1}). \tag{8}$$

2.2 Text Generation Strategies

To generate a full sequence of token ids and thus a document, the model must choose the next probable token id in the document using a text generation strategy. After choosing the next token id $\hat{w}$, where $1 \leq \hat{w} \leq |V|$, it is appended to the old sequence $w_1 ... w_n$ to form $w_1 ... w_n, \hat{w}$. Depending on the text generation strategy, the models generate different documents of varying quality. Each text generation strategy has different advantages and drawbacks. The next sections will introduce a few core sampling strategies and highlight the main advantages and drawbacks.

Greedy Decoding

Greedy decoding is the simplest of sampling strategies. The strategy always chooses the token id with the highest probability. This always yields the best next token id, but not necessarily the best token id overall.

Its main advantage is the simplicity. The strategy does not require additional state. Only the current probabilities are needed. This also leads to low hardware requirements, compared to other strategies.

As the current best next token id is not often the best token id overall, the strategy does not produce the highest quality documents. The next token id is chosen by the following formula:

$$\hat{w} = \text{argmax}_{w \in V} P(w_i \mid w_1 ... w_{i-1}). \tag{9}$$

Beam Search

Greedy decoding suffers from the problem of always choosing the current most probable token id, even if it is not the best token id overall. Beam search improves upon this by modeling the decoding process as a search tree. Instead of choosing only a single token

id, the strategy keeps k possible token id choices. These k possible token id choices are called **hypothesis**. At each time step the beam search uses the current k hypothesis to independently continue the generation to create $k \cdot V$ possible token id combinations. The strategy then keeps the top-k combinations as new hypothesis.

By modeling the decoding process as a search tree, beam search can choose token ids that are better overall. This leads to a higher quality generation.

But as beam search keeps k hypothesis and decodes $k \cdot V$ possible token id combinations, the strategy requires more memory and processing power.

Top-k
Top-k is a generalization of greedy decoding and a sampling strategy. Sampling strategies randomly choose the next token id from the probability distribution. Top-k chooses the next token id from the top k most probable next token ids. Hence, the model chooses a wider variety of token ids, leading to richer outputs.

As top-k is a generalization of greedy decoding, it has the advantage of being more powerful.

The main drawback is the loss of determinism. Lacking determinism may complicate interpretability methods [12].

Top-p
Top-k always chooses from the top k most probable token ids, even if most token ids are very unlikely and only few of the k token ids are probable. Top-p allows for more dynamic shapes of probability distributions, by only sampling from the top p percent of token ids [13].

Temperature sampling
Temperature sampling takes a different approach to top-p. It does not adapt to the shape of the probability distribution, but reshapes it. It divides the raw probability scores by a temperature τ:

$$y = \text{softmax}\left(\frac{u}{\tau}\right). \tag{10}$$

The softmax function normalizes the logits to the range $(0, 1)$, where the sum of all values equals 1. When dividing by $\tau < 1$ all values increase in total. After normalization this skews the probability distribution towards the most probable token ids. In contrast, when dividing by $\tau > 1$ the probability distribution is flattened. This leads to a more uniform distribution.

2.3 Controllable Generation

CLMs suffer from many problems, such as outdated knowledge, insufficient knowledge and insufficient capabilities. Most of these problems can be addressed by fine-tuning the model on further data, such as up-to-date information. The problem with fine-tuning is the need for a large amount of up-to-date data, which might not exist yet. In addition, fine-tuning is resource intensive and must be performed continuously, to stay up-to-date.

This section will showcase different ways to control the generation of the model, without the need to perform fine-tuning. The text generation is complex and there are many different strategies and components, which may be controlled. Guided generation will focus on strategies to control the prediction of a single next token id, while the chapter prompting strategies will focus on prompting techniques.

Guided Generation
The CLM generates a text by iteratively choosing the next token ids. The decision on which token ids is the next token id depends on two factors, as discussed in Section 2.1. It depends on the logits produced through the raw next token prediction function f_{rntpf} and a decoding strategy. Guided generation is a technique to modify the logits before the decoding strategy is applied. The logits are modified according to a set of domain specific rules. The rules not only have access to the current logits, but also to the previously generated text. This allows for complex rules.

Consider Example 3:

Example 3: Enforcing "yes" or "no" answers

A model is tasked with answering a “yes” or “no” question. The model should only respond with either of both words, without discussing any details or making small talk.

The rules for the guided generation are to set the logits of all token ids to $-\infty$, except when the token ids contribute to a “yes” or “no” answer.

Given the prompt “Is the sun shining?” and the previous token ids of the token “Ye”, the guided generation will set all logits to $-\infty$, except for the token id of the token “s”. In the next step only the EOS token will be permitted. Thus, the model will respond with “Yes”.

Guided generation has the benefit of allowing complex rules to be applied to the generation process, to allow a very tightly controlled generation.

The downside of guided generation is, that for complex scenarios, the rules may become very complex and hard to define. In these cases prompting strategies might be more appropriate.

Prompting strategies
Prompting strategies are a set of techniques to control the generation of the model externally. The model itself, or the model runtime itself is not modified. As such this technique is model agnostic. The general goal is to control the output through structured prompts. In addition, the model may be incorporated into external tooling to enable complex workflows.

The simplest form of control through prompting is to add instructions to the prompt. Models will follow the instructions, if the model was trained on instructions in a similar format. So called “instruction fine-tuned models” were fine-tuned especially on instructions, to improve instruction following capabilities. Non instruction fine-tuned models can still follow instructions, but then the instructions must be chosen carefully.

A useful instruction is to define the desired output format. For example, by specifying that the model should output the response in JavaScript Object Notation (JSON) format

or Markdown format, the model will use the specified format. Especially formats such as Markdown and JSON are preferred, as they are common formats and thus the model performs especially well on these formats. Specifying the desired format increases the predictability of the response, which allows automated processing of the response. Specifying to respond in an unknown format does not work. Steering the model does not raise the capabilities of the model, but raises the efficiency with which its capabilities are used.

If the model does not have enough knowledge to respond to a prompt, the prompt itself can be structured to supply the needed information. Given the model has the ability to understand the provided information, it can then act on the information. A popular technique is **few-shot prompting** [14].

Few-shot prompting involves providing the model with several examples in the prompt. The goal is that the model understands the provided examples to learn the pattern and apply it to its response. The learning is undertaken "in context" and not persisted outside the prompt itself [14]. The knowledge provided to the model is the format of the task and matching response.

Example 4: Few-shot prompting

Prompt:
Q: What is the capital of Germany?
A: Berlin
Q: What is the capital of France?
A: Paris
Q: What is the capital of Italy?

Response:
A: Rome

As can be seen in Example 4, the model prefixes the response with A: and does not provide additional information. The output format was not specified explicitly, but matches the format of the examples perfectly.

In contrast to providing knowledge about a desired response format, **context injection** adds important context into the prompt, which the model then uses to respond.

Example 5: Context injection

Prompt:
The capital of Germany was changed to Heidelberg.
What is the capital of Germany?

Response:
Heidelberg

Example 5 demonstrates how adding information controls the response of the model. This technique is very powerful, but comes with two strong restrictions (non exhaustive):
1. Finding relevant context.
2. Limited prompt size.

The first restriction is that the injected context must first be found. This is a hard search problem, as the information must be retrieved *before* prompting, before the intent of the prompt is known yet. In Example 5 the task of the CLM is question answering. Even though the general task is known, the software has no way to know the specific domain, such as "country capitals" beforehand.

The second restriction is the limited prompt size. Models are only capable of focusing on a prompt of limited size. Thus, token space is precious and may not be wasted. Providing the model with an up-to-date version of Wikipedia for every prompt oversaturates the prompt size limit instantly. The injected context must be *targeted* and *relevant*. The more *targeted* the context is, the fewer tokens are used.

As finding relevant context is a hard problem, several steps are needed to search for the context. Searching for relevant context may even involve the output of another CLM. Strategies that involve the model in a multi-step process are called meta-strategies. These strategies will be further discussed in the next section.

Meta-Strategies

Meta-strategies incorporate a model into an algorithm according to a wider strategy. These strategies shift the focus to viewing the CLMs as a building block to be used as needed. Several models with different strenghts could be combined into a meta-strategy.

Generally, the main benefits are ease of use and flexibility. Any model can easily be integrated into the meta-strategy and is replaceable. By combining several models into a multi-turn process, including external tooling, powerful behavior is possible.

In contrast, common drawbacks of meta-strategies are the added complexity towards the already complex text generation. In addition, calling models several times increases the runtime of the program, if used without care.

Meta-strategies are applied to e.g. remedy the model lack of up-to-date knowledge. A model may not have the necessary knowledge for a task, or the knowledge it has is outdated. In such cases the meta-strategy can supply up-to-date information and focus the model towards the problem.

Chain of Thought. Chain of Thought is one very central meta-strategy, as it significantly improves the reasoning capabilities of a model [15].

The key idea of Chain of Thought is to apply the divide and conquer strategy to the text generation process. Instead of the model trying to solve a complex reasoning problem in a single step, the meta-strategy divides the problem into several intermediate steps. The model then is able to solve the simpler intermediate steps. Each intermediate step is a thought the model contributes to the final solution. The sequence of intermediate steps forms the "chain" of thoughts [15].

Additionally, the model will use thoughts of previous steps to reason about the next step.

Tree of Thought. Tree of Thoughts is a generalization of chain of thought. Chain of Thought is only capable of performing linear decision-making [15]. Ultimately, performing

complex reasoning is a search problem and thus, linear reasoning is insufficient. The Tree of Thoughts meta-strategy generalizes the thoughts into a tree structure. Each intermediate problem is then a node in this search tree. The root of the tree is the initial problem. The goal of this search problem then is to find an accepted end node - a finishing thought [16].

By structuring the thoughts into a tree, common tree search algorithms can be applied to the problem. Various algorithms such as a breadth-first search or depth-first search can be applied [16].

In general, the Tree of Thoughts meta-strategy is more powerful than Chain of Thoughts. It is a very powerful and flexible strategy, which even allows following thoughts and then backtracking [16].

2.4 Transformers Library

Transformers is the name of the popular Python machine learning library developed by HuggingFace. It provides broad support for many different machine learning domains, such as computer vision, audio processing, multimodal models and *natural language processing* [17]. For each domain, the library enables simple training and inference of state-of-the-art models. The thesis heavily relies on functionality provided by this library, as it provides in-depth features to implement the more advanced features. It provides support for all techniques introduced in Section 2.1 and Section 2.2 [17].

As the library receives frequent updates, the thesis focuses on version 4.41.3[1]. In addition, the library is complex and enables many different use-cases through many abstraction layers. This section will focus on the CLM use-case, omit unneeded arguments and list parameters only available for CLM.

The structure of this section will mirror the structure of previous sections. It will start by introducing CLMs, then continue with tokenizers. Afterwards, the text generation functions are presented.

Using different text generation strategies with the Transformers library will follow. At last, implementing controllable generation will be shown.

Basic Classes

The Transformers library provides a set of classes to interact with models, tokenizers and configurations. The base classes are PreTrainedModel, PreTrainedTokenizer and PretrainedConfig. These classes provide the basic functionality to load pre-trained models, tokenizers and configurations. The classes are inherited by the specific model, tokenizer and configuration classes, which provide the functionality to interact with the specific model architecture. Instances of these classes are created through the AutoClasses abstraction. The AutoClasses abstraction allows for easy loading of models, tokenizers and configurations, without the need to know the specific class names. The abstraction guesses the correct class based on the provided model name [18].

[1] https://huggingface.co/docs/transformers/v4.41.3/en/index

HuggingFace supports a wide variety of code generation models[1], such as:

- codellama/CodeLlama-7b-Instruct-hf,
- deepseek-ai/deepseek-coder-6.7b-instruct,
- m-a-p/OpenCodeInterpreter-DS-6.7B,
- ise-uiuc/Magicoder-S-DS-6.7B.

Models and tokenizers support adding custom tokens at runtime. First the vocabulary of the tokenizer must be extended through the add_special_tokens method. Afterwards, the model must be resized through the resize_token_embeddings method.

Text Generation

After obtaining a model and a tokenizer, the text generation process can be performed by using one of the following three abstraction. The to be introduced abstraction build upon the previous abstraction.

The lowest level abstraction is the forward method of the model, which takes previous token ids and returns the logits for the next token id. The arguments to the function only control the provided output format and cache behavior [19].

The next abstraction is the generate method, which generates a whole sequence of tokens until a stopping criteria is met, such as the EOS token. The functions is provided by the GenerationMixin class, which CLM inherit from using multiple-inheritance. The GenerationMixin implements the different text generation strategies, which can be controlled through the provided parameters [20]. The generate function has the following signature:

```python
generate(
    inputs: torch.Tensor,
    generation_config: GenerationConfig = None,
    logits_processor: LogitsProcessorList = None,
    stopping_criteria: StoppingCriteriaList = None,
    **kwargs
    // ... omitted parameters
) -> Union[ModelOutput, torch.LongTensor]
```

Listing 1: Generating a sequence of tokens

Next to the provided input tensor inputs, the generation_config parameter contains configuration values to control the generation. Extra matching keyword arguments override keys in the generation config. Table 1 contains important configuration parameters:

[1]More code generation models can be found in the Bigcode models leaderboard. (https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard)

Table 1: Generation configuration parameters

Category	Parameter	Type	Description
Stopping Criteria	max_length	int	Maximum length of the generated sequence
	max_new_tokens	int	Maximum number of new tokens to generate. Overrides max_length
	max_time	float	Maximum time in seconds to generate
Generation Strategy	do_sample	bool	Whether to sample from the logits
	num_beams	int	Number of beams for beam search
Logits manipulation	temperature	float	Temperature for sampling
	top_k	int	Number of tokens to sample from
	top_p	float	Cumulative probability for sampling from
	repetition_penalty	float	Penalizes repeated tokens

The first parameters in Table 1 configure the first stopping criteria. The generation can be stopped when the sequence reaches a certain length, or when a certain time is reached.

The following diagram displays the different used generation strategies, depending on the provided parameters:

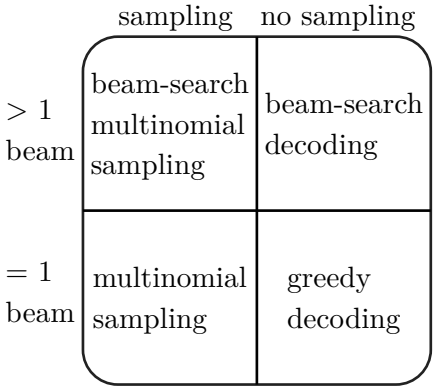


Figure 1: Generation strategies

The final parameters manipulate the logits before sampling. They enable Top-k, Top-p and Temperature sampling. The last parameter hinders models from endlessly repeating the same token, until a stopping criterium is reached.

Controllable Generation

The generation method provides parameters to overview and control the generation process. These parameters enable strategies as discussed in Section 2.3.

The first control mechanism are the stopping criteria. The stopping criteria define custom conditions that terminate the generation, when one of the conditions is met.

Transformers defines a few vital stopping criteria, like the maximum time or the maximum number of tokens to generate. These stopping criteria are enabled by passing certain keyword arguments into the generate method, as introduced in the section before [21, 20].

Custom stopping criteria are defined by subclassing the StoppingCriteria class and adding an instance of the new class to the stopping_criteria list in the generate method [21].

A subclass of StoppingCriteria only needs to implement the __call__ method:

```python
__call__(
    input_ids: torch.LongTensor,
    scores: torch.FloatTensor,
    **kwargs
) -> bool
```

Listing 2: Stopping criteria signature

The method in Listing 2 returns whether the generation should be stopped, based on the previous token ids input_ids and the current scores. The method must support batching, as input_ids has the shape (batch_size, sequence_length) and scores the shape (batch_size, vocab_size) [21].

The second control mechanism are the logits processors. Logits processors manipulate the current logits before a decoding strategy chooses the next token id on the then manipulated logits.

Custom logits processors are defined by subclassing the LogitsProcessor class and adding an instance of the new class to the logits_processor list in the generate method. A subclass of LogitsProcessor only needs to implement the __call__ method:

```python
__call__(
    input_ids: torch.LongTensor,
    scores: torch.FloatTensor,
    **kwargs
) -> torch.FloatTensor
```

Listing 3: Logits processor signature

The arguments in Listing 3 are identical to the arguments in Listing 2. The method must return the manipulated logits, which have the same shape as the input scores.

2.5 Language Server Protocol

Many modern editors and IDEs provide features such as:

- Autocompletion,
- Signature help,
- Go to the definition of a symbol,
- Go to the uses of a symbol,
- Provide diagnostics data on the current file,
- Rename symbols.

This list is not exhaustive, and the features between different editors may vary. These features are vital for any programmer, as the effort to create and maintain code is significantly reduced. Navigating large code bases and finding documentation on used code is highly beneficial.

These many features take a significant effort to implement for a *single* programming language and editor. Providing support for several programming languages massively increases the implementation effort. In addition, the created features must be maintained next to the editor itself. As more programming languages are designed and adopted, the need for tooling for these programming languages arises. Editor maintainers must then integrate these features for the new programming language.

Providing support for an ever-increasing amount of programming languages is hard, but supporting these features for an ever-increasing amount of programming languages in every editor is impossible. Many diverse editors exist, such as Emacs [22, 23], VSCode [24, 25], Neovim (modern fork of Vim) [26, 27], Zed (the successor of Atom) [28, 29], Jetbrains IDEs [30, 31], Helix [32, 33] and many more.

Thus, a solution is needed to extract the *common* features from the editors and provide them centrally, such that every editor need only implement an interface to integrate the common features. The Language Server Protocol is an effort to standardize a common feature set for programming language tooling for use in IDEs. The protocol was designed my Microsoft for use in its Visual Studio Code editor, but is now an open standard. This thesis will focus on version 3.17 of the protocol. The protocol includes many different features, of which the following sections introduce a few important. The full specification may be accessed here[1] [34].

Design Principles

The protocol is based on a bidirectional client-server architecture. The client is a tool such as the IDE. The server, henceforth called the language server, is programming language specific tooling that provides a subset of the protocol. Both the client and server must not implement the full protocol. The protocol is capability-based. Both parties define and communicate their capabilities, then find common capabilities and restrict their use to them. Each client can connect to multiple language servers, and each language server can serve multiple clients. As such a client could use a language server for the Rust programming language while the user is coding on a Rust project. As soon as the user switches projects to a web project, the editor could then connect to several language servers to provide support for HTML, CSS and JavaScript at once. In that scenario several simultaneous language servers are necessary, as CSS and JavaScript can be embedded into HTML. The communication between the client and server must be bidirectional, as the server must be able to notify the client about events occurring, such as new diagnostic information.

Communication

The base protocol relies on version 2.0 of JavaScript Object Notation-Remote Procedure Call (JSON-RPC). It is a light-weight and stateless protocol to encode Remote Procedure Calls (RPCs). The specification defines data structures and processing rules. As such, the protocol is transport-agnostic and can be used with any type of message passing framework. The data structures are encoded using the JSON data format.

JSON-RPC defines three types of messages:

- **requests**,

[1] https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#languageFeatures

- **responses**,
- **notifications**.

Each request by the client must be followed by a response by the server. Notifications are requests that do not require a response and may be sent by either the client or server. Each request contains a method field, which specifies the remote procedure to be called. The params field contains the parameters needed to call the remote function. All to be introduced language features have a method value, which when called, will trigger the feature.

Each raw message contains a header and content section, delimited by \r\n, where the content is a message as defined by JSON-RPC.

Lifecycle and Document Synchronization Messages
When interacting with a language server, the client must first initialize the server using lifecycle messages. After the initialization, the client can synchronize documents with the server. The server then provides language features for the synchronized documents.

The connection to a language server is initialized using the initialize request. During the initialization, no other communication is allowed. The client must acknowledge the initialization with the initialized notification. After initialization, files can be opened and closed using the textDocument/didOpen and textDocument/didClose notifications. The server then processes the opened files and provides language features for them. Finally, the connection can be closed using the shutdown and exit notifications. Further details on the lifecycle and document synchronization messages can be found in the LSP specification [34].

Language Features
This section displays messages to query language features. The LSP specification defines many different features, of which this section highlights the few relevant ones:
- Auto-Completion,
- Signature help,
- Provide diagnostics data on the current file.

The auto-completion feature provides a list of completion items for a given position in a document. The completion items can be selected to provide more detailed information. On applying the selected completion item, the completion text is inserted into the document at the given position. The feature is split into the two messages textDocument/completion and completionItem/resolve. The first message queries the completion items for the given position in a text document. Each completion item only contains basic information. If further details are required, the second message completionItem/resolve asks the server to extend a single completion item with in depth details.

```typescript
interface CompletionItem {
    label: string;
    insertText?: string;
    labelDetails?: CompletionItemLabelDetails;
    detail?: string;
    documentation?: string | MarkupContent;
    kind?: CompletionItemKind;
    tags?: CompletionItemTag[];
    deprecated?: boolean;
    preselect?: boolean;
    sortText?: string;
    filterText?: string;
    data?: any;
    // ... more fields omitted
}
```

Listing 4: CompletionItem structure

The first key property is **label**. The **label** is the main text to be inserted into the document and provides the full symbol. In case the document already contains a prefix of the **label** in the given location, the inserted text must not contain the prefix. In these cases the **label** property remains untouched, but **insertText** is set to the shortened text. **insertText** overrides the text to be inserted, but remains optional. The **labelDetails** property contains additional information, such as the locator of the symbol to be inserted. The structure of the property can be found in Listing 5. For accessibility, the completion item includes several human readable properties:

- **detail** provides a short description of the completion item.
- **documentation** provides a detailed description of the completion item. The description can be plain text or Markdown. The structure of the property can be found in Listing 6.

The next field **kind** specifies the type of the completion item. The type refers to the symbol kind, such as a "function", "field" or "string". The full list is defined in Listing 51. Annotations can be found in the field **tags**. The only supported tag is **Deprecated**. It replaces the now deprecated field **deprecated**.

After receiving the completion items, the client can display the items in a list. The field **preselect** can mark a **single** completion item to be preselected in the list. Additionally, the client may filter and sort the items. In these cases it should use the fields **filterText** and **sortText** for these tasks, instead of **label**. Additional server specific data is stored in the field **data**. This information is needed by the server to resolve the completion item and should not be changed.

```typescript
interface CompletionItemLabelDetails {
    detail?: string;
    description?: string;
}
```

Listing 5: CompletionItemLabelDetails structure

```typescript
interface MarkupContent {
    kind: MarkupKind;
    value: string;
}
```

Listing 6: MarkupContent structure

The Listing 6 structure includes markup content in the field **value**. The type of markup is defined in the field **kind**. Currently the two kinds **plaintext** and **markdown**.

To request completions, the client then sends the **textDocument/completion** (Listing 7) message. The request contains a **position** in the document **textDocument**. The optional **context** has further details on how the completion was triggered (**triggerKind**) and with which character it was triggered (**triggerCharacter**), as seen in Listing 8. Several trigger kinds are currently supported, such as invoked, triggered by a character, or re-triggered to expand an incomplete completion list.

```typescript
interface CompletionParams {
    textDocument: TextDocumentIdentifier;
    position: Position;
    context?: CompletionContext;
    // ... more fields omitted
}
```

Listing 7: Completion message parameters

```typescript
interface CompletionContext {
    triggerKind: CompletionTriggerKind;
    triggerCharacter?: string;
}
```

Listing 8: CompletionContext structure

The server then responds either with no result, directly with a list of completion items, or a completion list, as seen in Listing 9.

```typescript
interface CompletionList {
    isIncomplete: boolean;
    items: CompletionItem[];
    // ... more fields omitted
}
```

Listing 9: CompletionList structure

In addition to containing the completion items, the structure also may indicate that the list is incomplete and will be recomputed through the field **isIncomplete**.

Resolving a single completion item is done by sending the **completionItem/resolve** message. The message is the completion item to be resolved. The server then responds with the resolved completion item. The resolved completion item is a superset of the provided completion item.

The second feature is the signature help. The signature help provides information about the signature of the symbol at the specified position. A common use case is to display function and class parameters. Sending the **textDocument/signatureHelp** request to the

server returns the signature help. The request has a similar structure to Listing 7, as seen in Listing 10.

```typescript
interface SignatureHelpParams {
    textDocument: TextDocumentIdentifier;
    position: Position;
    context?: SignatureHelpContext;
    // ... more fields omitted
}
```

Listing 10: SignatureHelp message parameters

The context field contains additional information about the context in which the signature help was triggered. The server responds with the signature help in the structure of Listing 11:

```typescript
interface SignatureHelp {
    signatures: SignatureInformation[];
    activeSignature?: uinteger;
    activeParameter?: uinteger;
}
```

Listing 11: SignatureHelp structure

The symbol could have several signatures, of which each is represented by a SignatureInformation structure. Each single SignatureInformation has a label, to be shown in the UI, as well as the documentation regarding the whole item. Single parameters can be retrieved through the parameters field. When typing a function call, the currently typed parameter becomes the active parameter in the SignatureInformation (activeParameter). The parameters each come with a label and documentation field. The label either is a string, or a range referring to the label of the outer signature.

The signature help structure is defined in Listing 12, while the single parameter information is provided in Listing 13.

```typescript
interface SignatureInformation {
    label: string;
    documentation?: string | MarkupContent;
    parameters?: ParameterInformation[];
    activeParameter?: uinteger;
}
```

Listing 12: SignatureInformation structure

```typescript
interface ParameterInformation {
    label: string | [uinteger, uinteger];
    documentation?: string | MarkupContent;
}
```

Listing 13: ParameterInformation structure

2.6 Related Work

Several prior works pave the way towards enhancing neural code completion through the integration of additional context. The prior works use different strategies to control the

generation process. The first group focuses on improving the training process itself, to enhance future code completion. While these approaches are static and cannot adapt to specific dependencies, they serve as auxiliary steps in more dynamic approaches, by improving the general robustness of improved models.

The second group of papers embed additional context into the prompt, modifying the generation externally. This often leads to a simpler integration, allowing the approach to be used with even cloud-based model offerings, where there is almost no control over the generation.

Other papers modify the logits during the generation itself, increasing the flexibility even further, but integrating into the generation even tighter. This group can rarely be used with cloud-based model offerings, but is powerful due to the high level of control. The following sections describe prior works, grouped into these three groups. Beforehand, an introductory work is presented.

Introductory related work

The prior work "Program Synthesis with Large Language Models" by Austin et al. [35] is the oldest prior related work, being published in 2021. It serves as the basis for the following works, as it analyzes the limits of CLMs for program synthesis in general purpose programming languages.

One observation the paper presents is that the chosen sampling strategy is critical for good performance in code synthesis. Beam search performs extremely poorly, greedy decoding performs better and sampling performs best when allowing several samples. As the paper empirically determines, beam search often produces looped or repeated code [35]. The effect of a sampling beam search is not analyzed. The observations of Austin et al. mirror the results obtained by evaluating neural code completion using this approach.

The second observation stems from analyzing dialogue capabilities of models for code synthesis. The authors observe that natural language feedback from a human halves the error rate compared to the model's initial prediction [35]. These corrections often include small context errors, such as import and identifier errors. While the authors state that feedback in natural language can correct small context errors, can this feedback be automated by a machine? Can a program determine additional natural language feedback before or after generation and add it to the prompt? These questions serve as the basis for the prompt-based related works, which automate this step and use similar model capabilities to steer the models towards an improved code completion.

Training-based related works

The result of the work RepoFusion by Shrivastava et al. [36] is a framework for training models with improved capabilities of using code context correctly. The paper paints the scenario of a model performing code completion, in which it is unable to use the given code context correctly, as its ability of using correct Application Programming Interfaces (APIs) are low. The proposed framework uses a technique called Fusion-in-Decoder (FiD) to combine information from different code contexts (i.e. code files) into the training [36].

As RepoFusion controls the model training, the trained model still has limited knowledge during code completion, but can utilize the available knowledge even more so. As stated in the work, it is dependant on further tooling, which supplies the needed knowledge before

code completion, such that the provided knowledge may be used. While that leaves many problems unsolved towards using correct dependency information, it shows how model training may complement other approaches, which solve the tooling aspect. RepoFusion possibly lays the groundwork for complex future tools, such that they may fully utilize code context embedded into the prompt.

Prompt-based related works

The next group of papers focus on embedding additional information into the prompt (and could benefit from training improvements similar to RepoFusion). These related works propose meta-strategies, which use a strategy to retrieve additional information, which is then embedded into the prompt. The model is either used to directly generate a result, or used in a multi-step generation. First the single-step generation works are discussed, after which the multi-step works follow.

Single-step generation related works. The work DocPrompting by Zhou et al. [37] focuses on utilizing code documentation. It presents a general approach to retrieve documentation relevant to the code generation beforehand, embed it into the prompt and then proceed generating the result code. The documentation is sourced from online available documentation sites, such as DevDocs [38]. The retrieved documentation is then embedded. Relevant documentation is found by finding embeddings of documentation near the embedding of the prompt.

The benefit of such an approach is the high flexibility. It may be used with any model, as the only requirement is the regular generation interface. Additionally, documentation may be sourced from any source. Sites such as DevDocs may include additional documentation over regular documentation in dependencies, which may provide further high-quality documentation not available elsewhere.

The downside of DocPrompting is that official code documentation of dependencies is used indirectly. Instead of directly retrieving documentation of dependencies, it is sourced elsewhere. The sources must be kept up-to-date. Additionally, the paper only aims for the latest documentation, not considering scenarios in which models work on legacy code bases. Using embeddings of documentation, it is hard to distinguish between different versions of the same code documentation. The next downside is that searching for relevant documentation is based on the current prompt and not incomplete code during generation. The model is not provided with the information it requires *during* generation, but with information it may require due to the input. This problem will now be referred to as the "context retrieval problem".

The issue is that such information is unknown before the first generation. This leads to more complex multi-step generation approaches, or even approaches controlling the model during generation.

Nonetheless, DocPrompting demonstrates how integrating up-to-date code documentation improves code completion and the importance of code documentation for this thesis.

Multi-step generation related works. DocPrompting displayed the "context retrieval problem". Multi-step generation approaches remediate the issue by iterating on the result,

retrieving relevant context after generation, then restarting the generation with more accurate context.

RepoCoder by Zhang et al. [39] is one such work. At its core, the work is similar to DocPrompting. The context is retrieved using Retrieval Augmented Generation (RAG), based on a pre-built index. The RAG is based on the current version of the code and the result is embedded into the prompt. The main difference is that instead of indexing documentation, RepoCoder indexes all project files. The idea is for the meta-strategy to allow the model to properly utilize repository code and converge to an ideal solution.

The downside of such a work is clear. As the index only considers repository files[1], no other dependencies are indexed. Thus, RepoCoder cannot provide relevant context on a specific version of a dependency. Additionally, the solution to the "context retrieval problem" is slow, as multiple iterations are needed to arrive at a satisfactory code solution. As before, the upside is that by integrating the model generation into a wider strategy, this work can be used with cloud-based model providers.

A similar work is De-Hallucinator by Eghbali et al. [40]. It also uses an iterative generation to solve the "context retrieval problem". A difference is the index from which context is retrieved. Instead of indexing the project files before generation, the authors propose to index API references. They define an API reference as either a function, class or attribute reference [40]. These are extracted using CodeQL[2].

As both works are similar, both the benefits and downsides are the same.

The last work is IDECoder by Li et al. [41]. Compared to previous works, the authors identify that such RAG approaches suffer from too many returned results and proper filtering is necessary. The authors propose to use IDE-native tooling to extract relevant information. This is the first work in this group to leverage the LSP to extract information from code.

IDECoder uses a slightly different approach than the previous two works. The first difference is that static information is added to the initial prompt, such as docstrings, project paths, member functions, class attributes, etc., which are extracted from the project. An important note is how IDECoder processes dependencies. When encountering imports of third-party dependencies, the version is included as context, such that the model may differentiate between versions. The assumption is that the model has the capability to distinguish between versions as has enough knowledge about the third-party dependency.

The information is then merged and added to the initial generation prompt. The resulting code is then refined using a secondary process. The assumption here is that relevant context has been found and only other errors need to be corrected. Here the LSP is used to obtain diagnostics information about the generated code, such as mismatching types. The information is embedded into the prompt, resulting in a refined version of the code.

IDECoder showcases similar benefits as previous approaches. It can still be used with cloud-based model offerings, as the model is only used for generation. Additionally, high

[1] Only the persistent core project files are considered project files. This excludes temporary project environment directories, such as `node_modules` or `venv` directories.

[2] https://codeql.github.com/

quality context information is used through language servers, even providing automated human feedback on the code.

While the approach shows less downsides, it may still struggle when using specific versions of dependencies, as the approach requires the model to be trained on samples of the dependency. This is not sustainable and IDECoder does *not* use the language server to provide such accurate context information.

Logits manipulation-based related works

Previous works can be widely integrated into different runtimes an even be used with cloud-based model offerings. However, common issues include the slow runtime due to multiple generation steps and unrelevant context due to the "context retrieval problem". More recent works propose to control the generation process itself, by modifying the logits during generation. This allows for a more nuanced control over the generation process, but may require a higher level of integration with the model. Monitor-Guided Decoding (MGD) by Zhang et al. [3] is one such technique. The authors note that the model way be monitored during generation, such that relevant context may be retrieved and used ad-hoc. This allows for a quicker guided generation, as only a single generation step is necessary. The proposed approach combines a generation hook with static analysis through language servers with logits manipulation. The generation hook obtains the current incomplete version of the code, enabling online static analysis. MGD focuses on type-consistent object-dereferencing, guiding the model when generating a dereference operation on an object, such as accessing a member function or attribute. The generation is restricted to only allow available member functions or attributes of the type. Restriction occurs by modifying the logits of the current generation step.

In contrast to previous approaches, this work is more complex. It uses both static analysis through a language server and is operating during the generation itself. While this increases the overall complexity, it achieves higher speeds while accurately pinpointing relevant context. Using the language server even enables the model to generate type-consistent object-dereferences for types of any source, such as third-party dependencies [3]. MGD is the first listed approach to correctly provide accurate context information for third-party dependencies. While this is a great basis for further works, it fails to address the issue of code versioning, such as deprecated functions. When guiding the model towards an attribute of a type, no distinction is made between deprecated and non-deprecated attributes. This leads to models generating code using deprecated attributes.

Additionally, the scope of retrieved information is limited. By using a constraint-based guided generation, modifying logits, embedding information such as docstrings is hard. Simple guidance with little state between generated tokens is feasible, but shifting logit scores based on docstrings is not. As shown before, guiding the model using docstrings is more easily done through the prompt. A hybrid approach could leverage a language server to integrate textual information into the prompt, then further guide the model during generation using other context.

3 Approach

After introducing the necessary background in Chapter 2, this section will first derive the requirements from the motivation and goals in Section 3.1.

By analyzing the shortcomings of existing solutions in Section 3.2, the thesis will propose a new approach to integrate high quality context into neural code completion. Context retrieval is enhanced with handcrafted filtering rules, to reduce noise and improve completion accuracy.

Lastly, a dataset is introduced to evaluate the effectiveness of the new approach.

3.1 Requirements

The general requirements of this thesis are derived from the motivation and goals, namely the approach of a Copilot like solution which has *full* knowledge of the local development environment.

The general requirements in descending order of importance are seen in Table 2.

Table 2: General requirements of the thesis

Identifier	Keyword
R1	Dependency Accuracy
R2	General Correctness
R3	Robustness
R4	Generalizability
R5	Performance

Dependency Accuracy
Neural code completion works under the assumption that the CLM has knowledge of the used code dependencies. This requires either vast amounts of training data on code written using a similar version of the dependencies or the local code to already use the dependency correctly. The first option has the problem, that the training data is not annotated with the correct dependency version, thus CLMs cannot differentiate between different versions of the same dependency. The approach works well for widely used dependencies, which have breaking changes rarely and do not have many different used versions. In contrast, the approach does not work for dependencies which are either too new, or too unused to be known to the CLM. Another problematic type of dependency is the often changing dependency, as the model cannot process breaking changes in the dependency.

The second option, in-context learning, allows the CLM to dynamically learn a small subset of the dependency. The benefit of such a dynamic approach is that the CLM has accurate knowledge of dependency knowledge previously unknown. The huge flaw is that the CLM can only copy the developer, thus only achieving quality on par, but not above the developer. Additionally, the developer has to write the initial code, in which case the developer either misuses the dependency or already acquired knowledge of the dependency, in which case the CLM cannot support him in that regard.

1. The proposed approach shall provide neural code completions with a high accuracy, by always considering the *exact local development environment* and the full feature set of each dependency.
2. It shall differentiate between nuances of different versions of the same dependency.
3. To enable diverse code completions, the approach shall be able to handle a wide range of dependencies, even if they have never been used in other public projects before.

The second dependency accuracy requirement has the implication that the approach should follow (some) clean code practices. Dependencies use a "deprecation" mechanism to gradually change features, without breaking dependent code directly by providing a time window, in which the code can be changed, before the old feature is removed or replaced. By differentiating between nuances in different versions, the approach can suggest code which is future-proof and does not rely on deprecated features. The following different scenarios between dependency versions can be distinguished:
1. The dependency has a new feature, which is not available in the old version.
2. The dependency has a feature, which is deprecated in the new version and will either be removed or replaced.
3. The dependency has a feature, which is not available in the new version.

It should be noted, that local code bases are a special case of a dependency and should not be treated differently, as many different works do. Locally available code is also imported and used alongside remote dependencies. As such, the *scope* of this thesis includes prediction of all possible code.

General Correctness
The proposed approach builds upon other works to improve especially dependency handling in neural code completion. Still, the basic assumptions on code correctness should still stand. In particular, the approach should suggest syntactically correct code. Additionally, the approach should suggest code that is functionally correct and solves the given task.

Both correctness requirements are the essentials, which are expected from any code completion tool, upon which the thesis improves. The focus of this thesis lies on improving methodical correctness through Dependency Accuracy and functional correctness and General Correctness are secondary.

Robustness
Developers may work on code in a variety of different environments and using different types of dependencies. The approach should be robust under any circumstances and still provide reasonably good completions.

Specifically, the approach should be able to handle the following scenarios:
1. The size of the project should be irrelevant. The approach should scale to any project size.
2. The approach should be able to handle any type of dependency, irrelevant of the size or complexity of the dependency. It shall not matter if the dependency has no documentation or has pages full of documentation.

Generalizability
The programming language ecosystem is vast and diverse. Even though only a few programming languages are widely used [42], the approach should be able to handle most

programming languages. By supporting lesser used programming languages, this approach can be used to help developers in a wider range of projects, such as hobby projects. It could be even used to maintain legacy code.

Performance
Using editor tooling feels fast, as responses in interactions with toolings only take a few milliseconds, up to a second. Integrating CLM for code completion increases the latency by a significant margin, as inference takes several seconds up to a minute.

While the goal of this thesis is not to reduce the latency in neural code generation, the approach should not impose a significant overhead on the default code completion.

3.2 Analysis

After introducing the five requirements in the previous section, this section will analyze the shortcomings of existing solutions in regard to the requirements. In particular, the analysis will focus on the work by Agrawal et al. [3], as it is the most promising work. While this thesis has evolved without the knowledge of the work by Agrawal et al. and is not based on it, the general design choices are similar enough to show initial observations and design choices. Additionally, the term MGD is borrowed from that work, as it is a fitting name for the method it represents.

Monitor-Guided Decoding
In short, the work by Agrawal et al. fulfills all requirements except the Dependency Accuracy, but is limited in scope when considering dependencies.

By guiding the neural code completion selectively through the monitor, the approach improves upon the default code completion and fulfills the requirement of General Correctness. Meanwhile, the Dependency Accuracy is only partially fulfilled. The work takes an important step in using the LSP, as it enables retrieving context globally of the exact version of used dependencies. As the work only guides the dereference operation and function arguments count, the approach does not consider changes between dependency versions, such as deprecated items. It cannot guide the model away from now deprecated features.

Additionally, other code items, such as function signature parameters are not considered in the work.

As the work depends on the LSP, it is robust under the assumption, that the running language server is robust. By abstracting language dependent features, the work is generalizable to other programming languages, though the implementation of the work relies on Java specific functionality. Through the use of MGD, the only performance overhead of this method is the additional processing for each token. The bulk of the processing is shifted towards the language server, which performs the static analysis. As the language servers are optimized towards these analyses, the performance impact is $O(n)$, where n is the token count [3].

RepoCoder
The work by Zhang et al. fails in most requirements, only fulfilling basic requirements. It improves the default neural code completion through the use of repository level code snippets, thus fulfilling the General Correctness requirement. But in only considering repository level code, it fails to work on remote dependencies and cannot capture the nuances

between different versions of dependencies. Thus, it also fails the Robustness, especially as embedding of the local repository must be performed upfront. This upfront embedding of local code scales with the repository size and taking a considerable time for larger projects. By reusing local code, the approach is generalizable to other programming languages, as the retrieved context is of the same language. Under the assumption taht the embedding is performed upfront, the performance decreases by the iteration count i.

DocPrompting
The work by Zhou et al. fails similarly to RepoCoder, as it uses a similar approach, but tuned towards globally available documentation. As no iterations are used, the performance impact is a single similarity search using embeddings.

Summary
While DocPrompting and RepoCoder show that using documentation and available code can improve neural code completion, they fail to improve completions for very specific installed dependencies. The work by Agrawal et al. fulfills the most requirements, through the clever use of LSP. LSP is a protocol abstracting programming language features, thus enabling generalizable approaches using global context of the exact installed dependencies.

A possible approach would use available information, such as the CompletionItemTag in a CompletionItem, to check if an item was deprecated and should not be used anymore. Furthermore, additional language features, such as signature completions, could be used to guide the neural code completion of parameters in function calls.

A full MGD is not possible, due to deprecated completion items. A typical MGD approach would keep the logits of completion item to guide the dereference operation, as seen in Agrawal et al., but mask other tokens.

When combined with deprecated completion items, tokens of deprecated completion items would need to be masked, *while* tokens of non deprecated completion items would not be masked. The problem arises when deprecated and non deprecated completion items share the same prefix when represented as a token sequence. Now the same token would need masking and retaining.

The thesis will thus propose an alternative approach, which enables the use of both types of information sources, by reranking logits of the CLM locally in the monitor and globally by altering the prompt with injected information. The approach is threefold and includes a novel meta-strategy to guide the model through prompt-engineering, with dynamically obtained information from static analysis. The second part is the approach based on the first novel meta-strategy. Lastly, the approach of a new dataset is introduced, to evaluate the performance of the proposed approach.

3.3 Interrupt-Based Meta-Strategy

As sketched in Section 1.3, the Interrupt-Based Meta-Strategy (IBMS) is a novel approach to integrate context into model generation. It is heavily focused on integrating multiple information sources independently. The context shall be integrated seamlessly, without affecting the generation flow of the initial generation, except providing the additional information. The strategy shall serve as a basis for globally guiding the model generation, by dynamically injecting relevant context into the prompt.

Requirements

The interrupt-based meta-strategy is just a single part in the overall approach. Thus, the requirements are separate from the general requirements.

The exact requirements are listed in Table 3.

Table 3: Special requirements of the IBMS

Identifier	Keyword
$\mathbf{R_{IBMS}1}$	Multiple source support
$\mathbf{R_{IBMS}2}$	Flexible context location
$\mathbf{R_{IBMS}3}$	Minimal overhead

Multiple source support. The goal of the strategy is to integrate context of multiple sources. As such, the strategy must be able to differentiate between different sources and process these independently.

Flexible context location. In addition to integrating multiple sources, the integrated context should be able to be placed anywhere in the prompt. It could be integrated into the system prompt, the user prompt or even the model output at "runtime".

Minimal overhead. The last requirement is for the strategy to have minimal overhead over the default model generation. Integrating context should not affect the generation flow of the model in any way, not even performance wise.

Heritage

While previous sections introduced the need for this strategy, this section will discuss the observations leading to the design of the strategy.

While the strategy was developed independently of Agrawal et al. [3], the design uses the same observation made for the Monitor component in their work. The Monitor not only allows controlling model logits at runtime, but it also allows **observing the generation in real time**. This allows a fine-grained control over the model generation, which is not possible with general prompt engineering or meta-strategies. Coupled with any form of static analysis, the monitor can find additional relevant context. This is similar to the iterative approach of RepoCoder. RepoCoder uses each iteration to retrieve relevant code based on the current whole generation, while this observation leads to an approach which finds context during a single generation.

The second observation is that the Monitor is only able to influence the generation on a token by token basis, but cannot guide the whole flow of the generation. Meta-strategies and prompt engineering can guide the whole flow of the generation, but cannot retrieve relevant information efficiently.

The third key observation bridges the gap between meta-strategies and fine-grained control from the Monitor. Additional tokens can be added to the vocabulary of a tokenizer for inference and the model embeddings can be resized to fit the new token. The model will

not be able to meaningfully predict the added token, as it was not trained on the ad-hoc token, but strategies can use the added token to elicit behavior. For example, the token can be used as an additional EOS token, stopping the generation process, but with the added semantic information of the special token.

Combining the aforementioned observations, the IBMS combines the fine granularity needed to monitor the generation in real time, with the prompt editing capabilities of multi-step meta-strategies, through the use of an added special token.

Components

An overview over the strategy is given in Figure 2. All components in blue are additions to the normal model generation.

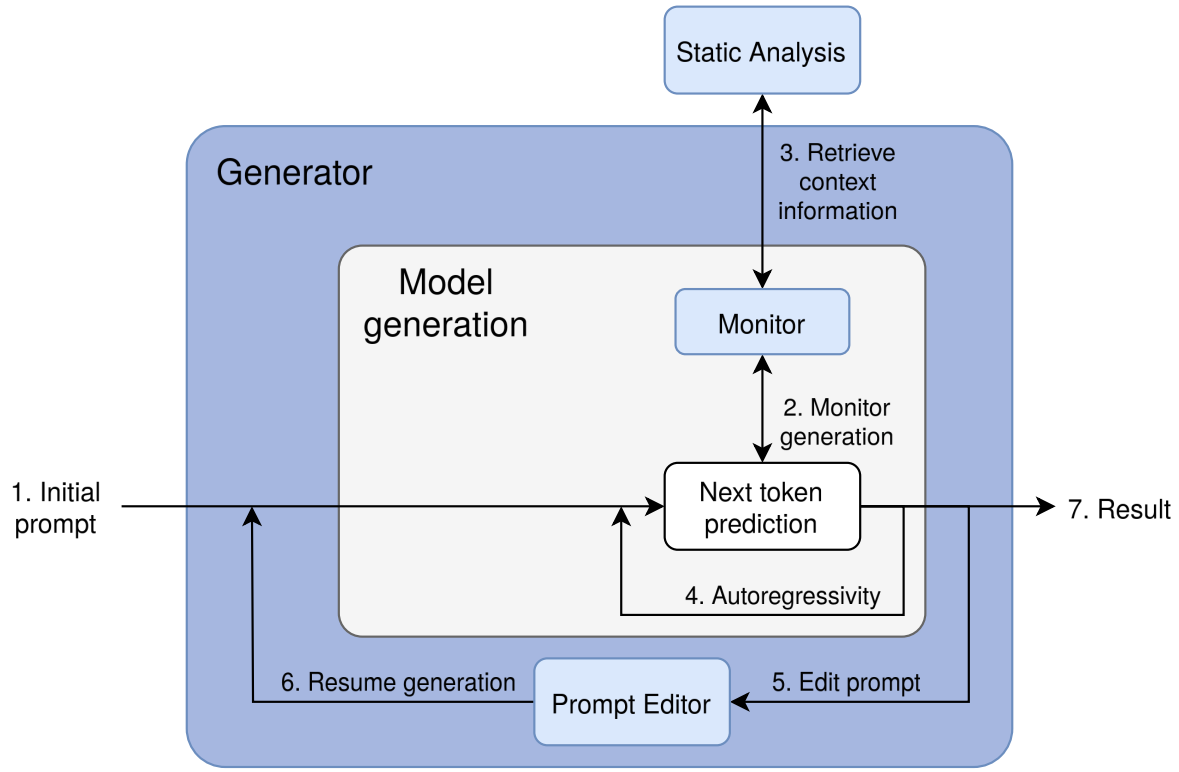


Figure 2: Components of the IBMS

In summary, the components **Generator**, **Prompt Editor**, **Monitor** and **Static Analysis** allow context interruption through a custom interrupt special token, prompt edits and resumption through a controlled model generation loop.

The first main component is the **Generator**. It is the main component initializing and orchestrating the different components. To generate a response using the IBMS, first a **Generator** must be initialized. It in turn will initialize the **Monitor**, the **Prompt Editor** and a connection to external **Static Analysis** tooling.

In addition, it will initialize the tokenizer and model, adding a custom special token to the vocabulary of the tokenizer and resize the model embeddings to fit the new special token. The new special token is called the interrupt token and is only used during the approach. No further training with the token is necessary, as the model will not need to predict the tokens, as the **Monitor** component will force the prediction of the interrupt token.

The Generator will start the model generation process, providing the initial prompt to the model. After the model finishes the whole generation, it will check the final token. If the final token is **not** an interrupt token, the generation process finished normally and the final generation result can be obtained by removing any remaining injected context through the Prompt Editor. Else the generation was interrupted with additional context and the Prompt Editor can edit the prompt with the context. To allow the continuation of the generation, the final interrupt token is removed from the generated token sequence.

Then the next component, the Prompt Editor is called. It will read the retrieved context and edit the prompt in any way. To edit the prompt, the Prompt Editor must work on the textual prompt, by detokenizing the prompt, applying any edits and tokenizing the prompt again. To perform edits such as deletions, the Prompt Editor must be aware of the location and type of inserted context. Thus, it stores a list of injected context, each with their location and type.

To inject context, the editor inserts the textual representation of the context at a location, then saving the location and type of the context. If the location is before any other injected context, the locations of the following contexts shift forward by the length of the textual representation of the injected context and must be changed accordingly.

Removing context is done similarly, by finding context in the list of all injected contexts, removing it from the stored location and shifting the location of all following contexts back by the length of the removed context.

```
[INST] <<SYS>>
system prompt
<</SYS>>
user prompt
[/INST]
model output </s>
```

Figure 3: Format of a typical model generation result

Figure 3 shows the format of a (partial) model generation output. The editor is flexible and can edit the prompt in any way, even inserting context into the partial model output.

The edited prompt is then passed into the model generation again, repeating the generation process.

Inside the model generation, the component Monitor watches the generation process, deciding if an interrupt is necessary. It was injected into the default model generation process by the Generator. The Monitor is called for each token prediction, receiving the previous token sequence and the next token prediction and optionally modifying the prediction logits. In its functionality it is similar to the monitor in Agrawal et al. [3], but with the difference that it only controls the prediction of the special interrupt token. If an interrupt is necessary, the Monitor will force the prediction of the interrupt token. In addition, it will store the retrieved context for access by the Generator and Prompt Editor after interruption.

To decide if an interrupt is necessary, the Monitor will call an external Static Analysis tool. The Static Analysis component represents any form of external tooling that can analyze the current generation and provide additional context. This could include a web search of named entities, a keyword search in local documents or LSP tooling. If no additional context was found, the model generation proceeds normally. Next to deciding if an interrupt is necessary to add context, the Monitor may also decide to remove old context. The interruption process is similar, with the difference in the stored context information.

To reduce the number of interruptions, the Monitor will only interrupt the generation if the additional context is not yet present in the prompt. To allow interruption of the model generation, the Generator adds a stop condition to the model generation loop. The stop condition checks for the existence of the interrupt token in the generated token sequence. As soon as the Monitor forces an interruption, the stop condition stops the model generation.

The IBMS adds a new loop to the model generation process, the interrupt loop. While the generation loop encompasses steps 2-4 in Figure 2, the interrupt loop additionally encompasses the steps 5-6.

Generation Strategy Support
The design mentioned in the previous section uses multiple model generation steps to arrive at a final result. This is similar to a typical multi-step meta-strategy, but with the difference that no major changes are made to the prompt. Each prompt to the model is an edited version of the previous prompt, with the only difference being the injected context.

Moreover, the strategy has a tighter control over the model generation, through the use of the interrupt token *and* support for model generation strategies. The previous section simplified the resumption of the model generation behavior, suggesting that the model generation can be resumed directly after interruption and context injection. This is the case for simple decoding strategies like greedy decoding, which generate on a single sequence.

Complex decoding strategies like beam search require complex handling, as they operate on multiple sequences. At the start of a typical generation process using beam search, firstly k hypothesis of the input prompt are initialized. Then the generation process is started, with each hypothesis predicting the next token to form $k * V$ combinations, of which the top-k combinations are kept.

When the stop condition is met, the *singular* top hypothesis is selected as the generation result. The Generator must ensure that the interrupted generation returns the current top-k hypothesis and is resumed with the edited top-k hypothesis.

To support complex decoding strategies, the Generator must use an edited model generation loop, that can store all sequences currently in the generation process and which can resume the generation process with existing sequences. Additionally, interrupts occur on a single sequence, not affecting other sequences. Each component storing state must store the state for each sequence separately. E.g. the Prompt Editor will store the injected context for each sequence separately.

Furthermore, the Generator must ensure the correct order of sequences, as the order of sequences can change during the generation process, e.g. in the top-k selection process in the beam search, as it may keep the top-k sequences sorted descending. After interruption, the then current top-k sequences originate from $l \leq k$ original sequences, where l depends on the spread of selected sequences. If each of the k sequences is continued at each step, then $l = k$. If the top-k sequences all originate from a single sequence, then $l = 1$.

3.4 Language Server Protocol aided generation

The LSP aided generation builds upon the IBMS. It applies the strategy to neural code completion, injecting context information from the LSP server into the prompt at runtime. In addition to injecting context into the prompt, the logits in the next token prediction are controlled using the context, to improve the robustness of the work.

This section will first list the thoughts leading towards the design of the work. Then, it will provide an overview of the components introduced in the work. Finally, it will detail the design decisions made for the components.

Heritage

The key thought guiding the work is:

> Developers base their decisions on contextual information provided by the LSP server.
> Why can the neural code completion not do the same?

Developers use a wide variety of tools to guide their programming decisions. Code completion is the most frequently used tool inside IDEs [1]. It displays available symbols, their documentation and type information. Next to code completion, signature help is another important tool, as it displays available parameters of selected functions and their types, thus providing information about the structure of the data. Additionally, IDEs highlight the code with diagnostics information, such as important errors or warnings which may be ignored. The diagnostics information guide the developer to potential problems, such as wrong symbols, code smells, or dead code.

This approach aims to guide the CLM using the same information sources developers have access to. The analysis in Section 3.2 already highlights the benefits of using a language server in combination with a MGD-like strategy. Section 3.3 provides insights into how these information sources can be integrated into a neural code completion model.

The next observation is that the developer is provided with a vast amount of information, of which only a few select items are relevant. The approach must perform heavy filtering to provide only the relevant few items.

The last observation is that CLM may be heavily biased towards generating old deprecated code, even when guided away using prompt injection. As such, the approach must force the model away from deprecated code by directly manipulating logits.

Components

An overview over the work is given in Figure 4. All components in blue are additions to the normal model generation.

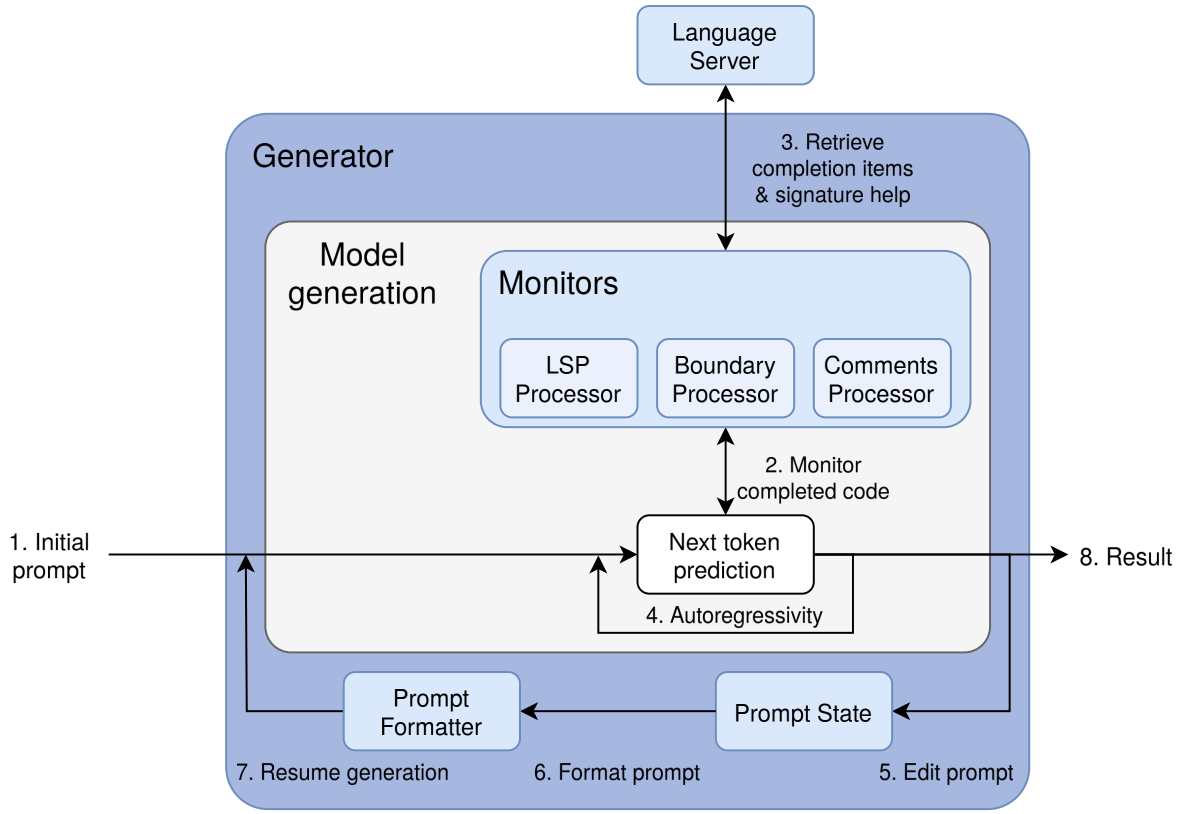


Figure 4: Components of the LSP aided generation

As the work is based on the IBMS, the components are similar to the ones in Figure 2. The main difference is the added complexity in the previous **Monitor** component, which now features several distinct subcomponents. Other differences are the specific use of LSP and a more complex prompt processing.

Generator. The **Generator** component depicted in Figure 4 is more specific in handling the prompt and injected context. The previous component **Prompt Editor** is now replaced with the two more specific components **Prompt State** and **Prompt Formatter**.

Prompt State. The **Prompt State** component contains a structured representation of the initial prompt and all injected context information. It is a general utility class, which is used in the **Monitors** and in the **Generator** to extract the completed code and to reconstruct the edited prompt. Extraction of the completed code is possible by detokenizing the partial result and removing the initial prompt prefix. As often the detokenization is not the inverse of the tokenization, the **Prompt State** class includes fixes for the detokenization. To reconstruct the edited prompt, the **Prompt State** passes the initial prompt and injected contexts to the **Prompt Formatter**.

Prompt Formatter. The **Prompt Formatter** abstracts different prompt formats used in different models. For models based on Llama, it would use the format defined in Figure 3. Different models allow different prompt sections, such as a system and a user prompt. Depending on the availability of such sections, the component decides how to embed the context.

LSP Processor. The difference in the Monitors is the threefold divide into subcomponents. The first subcomponent LSP Processor corresponds to the previous component Monitor. It receives context from a Language Server (previous Static Analysis) and enables the IBMS applied to code completion. The task of the LSP Processor is twofold. First, it enables the IBMS. Secondly, it controls the logits to force the model towards certain behavior, when guiding is not enough.

Boundary Processor. The purpose of the Boundary Processor is to fix boundary issues when using tokenizers which allow multi-character tokens. Usually, LSP information, such as autocompletion and signature help, is requested after every keystroke in an IDE, or for certain hotkeys. The autocompletion information is especially useful after the dereference operator (. in Python, -> in C++), as then available symbols are displayed. The signature help information is useful after starting to write a function call, as then the available parameters are displayed.

As tokenizers allow tokens with several characters, the vocabulary could include tokens that start with a trigger character, followed by any tail. For autocompletion, this could include tokens such as ".model". For signature help, this could be a token such as "()". By predicting these tokens, the completed code would "jump" over the critical trigger characters, thus ignoring the autocompletion or signature help information.

The Boundary Processor fixes this issue by identifying all tokens prefixed with a trigger character and replacing these tokens with the trigger character only. It ensures that the logits value of the trigger character is higher than any other token containing the trigger character as a prefix.

Comments Processor. Next to the Boundary Processor, the Comments Processor solves an issue regarding the generation of comments. A possible place to inject the context is in the completed code, as a code comment. The problem is that a model could see the code comments after an interrupt and try to predict a code comment preceding the next code block. Thus, code comments could be tainted by the code prediction itself. The Comments Processor disallows the generation of code comments, by stopping the prediction of the token starting a code comment by setting the logits value to $-\infty$.

Design Decisions

This section provides details on different design decision for the introduced components. The first design decision will be for how the IBMS is integrated. Then the different processing steps will be detailed. At last, the reranking details are presented.

IBMS Integration. The first design decision is the selection of sources to suggest fitting code. The previous sections already highlight why the LSP is fitting. The protocol provides a vast amount of different information. We argue that the core information to use are the completion items and the signature help information. When generating code, the primary source is the completion items, as they dictate the available symbols. The completion items list the attributes of objects and which attributes are deprecated and should be

replaced. After using the completion items to write callable code, the secondary source is the signature help. Likewise, the signature help lists available parameters of a function call and includes documentation. Both types of information sources allow code generation using up-to-date information. Other features provided through LSP may improve code generation, but are not vital, such as diagnostics information. In addition, both sources can be used on partial code and provide suggestions on how to extend the code, aiding the natural CLM generation flow. Diagnostic information and other information reference code already written, increasing the difficulty to integrate into the natural generation flow.

The second design decision is about the position of the injected context in the prompt. We argue that this position is dependent on the instruction following capabilities of the model. The used CLM must be able to react to the injected context. For most instruction tuned models, the context should be injected into the user prompt, to fit the model's capabilities. If the model is not instruction tuned, then the context should be injected into the generated code as a code comment, to mirror code "instructions" found in live code. This leads to the need for a **Comments Processor** to suppress the generation of normal code comments.

Information Filtering.

This section details the different processing steps for the retrieved completion items and signature help. Both sources provide critical information, but also a lot of irrelevant information not needed to decide on the next code piece. Such context pollutes the context window of a CLM and worsens the prediction quality. As such, preprocessing is needed to filter such data.

Completion items are initially filtered by their **CompletionItemKind**[1]. Only items of kind **Method**, **Field**, **Class**, **Function**, **Property** or **Variable** are allowed, to remove kinds such as **Snippet**, **Color** and **File**.

To reduce confusion further, completion items of builtins are removed, as the assumption is made that the model has the knowledge of builtins. If the model does not have knowledge about builtins, this filter can be disabled, to e.g. improve support of low resource languages. Filtering is done by checking the **detail** field of the completion item, as it contains the origin of the code item, such as the code file.

Next, completion items from the current code piece are removed. Similarly, it is assumed that the current code is in the context of the CLM. Again, this filter checks the **detail** field of the completion item.

Then private or meta functions are removed, such as the **__getitem__** function in Python. These functions are not meant to be called directly.

To avoid the completion of already completed code, the to-be-inserted text is checked against the ending of the currently generated code. This also removes items that do not properly extend the current code.

[1]The full list is visible in Listing 51

Then the completion items are split into two categories, the non-deprecated and the deprecated completion items. This differentiation allows later steps to guide the model towards non-deprecated items, while avoiding deprecated items.

The model may be biased towards generating old deprecated code. To only focus on the deprecated items, which the model *may* predict, the deprecated items are finally filtered by the next predicted token id.

Logits Manipulation. These design choices try to ramify the problem of overlapping tokens in deprecated and non-deprecated completion items. Instead of masking all token ids that are not a prefix of one of the completions items, this argues for the following:

- Token ids that are a prefix of a completion item should be scored higher.
- Token ids that are a prefix of a deprecated completion item and a non-deprecated completion item should be scored lower.
- Other token ids should be masked.

The exact delta score is not specified by the design and is left to the implementation.

The score of the interrupt token is fixed to the maximum possible logits value, if an interrupt should occur.

3.5 Dataset

Evaluation of the approach described in Section 3.4 requires a fitting dataset. This section will first list the requirements for such a dataset, then shortly analyze why current datasets do not fit these requirements. Lastly, it will propose the hand-crafted dataset DependencyEval that fits the requirements.

Requirements

In general, such a dataset requires a set of code snippets that can be used as input to the approach, as well as a ground truth for comparison. Moreover, it needs test cases to run the comparison and an evaluation pipeline to run a full evaluation securely and collect metrics.

As the work in this thesis focuses on exact use of dependencies, it has further requirements towards a dataset. The exact requirements are listed in Table 4.

Table 4: Special requirements of the dataset

Identifier	Keyword
$\mathbf{R}_{\text{dataset}}\mathbf{1}$	Captures exact environment
$\mathbf{R}_{\text{dataset}}\mathbf{2}$	Test functional and approach correctness
$\mathbf{R}_{\text{dataset}}\mathbf{3}$	Include examples of deprecated code
$\mathbf{R}_{\text{dataset}}\mathbf{4}$	Secure evaluation pipeline

Captures exact environment. The first requirement is that each entry in the dataset lists all details that constitute the environment in which the code is executed. In a broader sense, this requirement starts with the exact version of the interpreter / compiler used

to run the provided code. Then, the used dependencies need to be pinned to a specific version. Standard library packages often do not need to be pinned, as they are part of the language runtime and share the same version. In contrast, all external dependencies need to be specified.

Only specifying the dependencies is not enough, as this enables the reproduction of the environment but does not guarantee that it is usable. To ensure that the then available dependencies can be used, the import statements are required.

Test functional and approach correctness. The second requirement is for the dataset to contain at least two types of tests. The first type is used to test the functional correctness of any given code snippet. This test is the most basic and comparable to a regular unit test checking the output of a function.

The second type of test is used to check if the approach used in the code snippet is correct. This test will be the focus of the evaluation as it specifically checks if the code snippet e.g. uses deprecated code or uses the correct alternative instead.

Both tests can be run in conjunction to count the number of failed tests, completed tests, and tests with errors. This will allow observing the behavior of different changes in the implementation of the used approach while asserting that the functional correctness is kept.

Include examples of deprecated code. The third requirement is pertaining to the type of collected code snippets. To evaluate CLMs correctly using dependencies, the dataset requires examples of changing dependencies.

As noted in **R1**, there are three scenarios for dependency changes:
1. A feature is added to a dependency.
2. An old deprecated feature is removed or changed in a dependency.
3. A feature is removed in a dependency.

In addition, the included examples should cover the following other scenarios listed in **R1**:
4. A widely unknown dependency is used.
5. A dependency with a widely unknown feature is used.

The last two scenarios cover the case that the number of examples of the dependency or the specific feature in the training dataset of a CLM are too low to be learned correctly.

Secure evaluation pipeline. As stated above, each dataset requires an evaluation pipeline to automatically run the different tests and collect the results. The base assumption is that the pipeline is robust under any circumstances and can stably run unsupervised for many hours. In addition, the pipeline will test arbitrary generated untrusted code snippets. This requires a secure environment to run in.

The secure environment must be able to restrict the amount of compute available to the pipeline, as well as other resources like memory and disk space. To terminate the pipeline

in a more predictable time frame, the pipeline should terminate test runs taking too long, to prevent infinite loops or other problems.

While adhering to the security requirements, the pipeline needs to be able to create the exact environment for each code snippet, as described in $\mathbf{R}_{\text{dataset}}\mathbf{1}$.

Analysis
Several widely used datasets for code generation exist, such as MBPP, HumanEval, and ClassEval. The following will analyze these datasets towards the above requirements and note important findings.

MBPP. Mostly Basic Python Problems (MBPP) contains, as the name implies, mostly simple Python code snippets. It has around 1000 rows of data, each containing a problem statement, a working code solution and test cases to check the functional correctness [35].

It does not specify the exact environment in which the code is executed, nor does it need to, as it only contains simple Python code snippets that can be run in any modern Python environment. The only dependencies are 9 different Python packages included in the standard library. As the dataset is most basic, it is not focused towards the requirements of this thesis.

Neither does it contain tests for the approach correctness, nor does it contain examples for the listed scenarios in $\mathbf{R}_{\text{dataset}}\mathbf{3}$, nor does it contain a secure execution environment.

HumanEval. HumanEval is a dataset going slightly beyond MBPP in complexity. It contains 164 handwritten Python programming problems for code completion. Each programming problem has the scope of a single function and includes the function head with the docstring, the solution and test cases for the functional correctness. The solution is the completion of the function head and docstring and they can be combined to form the complete function [43].

Similar to MBPP, HumanEval lacks the same requirements. It only imports 8 packages, all of which are part of the standard library. In contrast, HumanEval contains a evaluation pipeline, but it only secures the execution rudimentarily, through the addition of a reliability guard.

The reliability guard wraps the to be executed Python code and removes functionality of the programming language environment that could be used to perform unwanted actions. As the securing does not happen on a system level, it only serves as a quick fix unsuited to rigorous evaluation.

Next to MBPP and HumanEval, the dataset EvalPlus has recently gained popularity, as it contains a more rigorous evaluation framework and includes MBPP+ and HumanEval+, which extend the aforementioned datasets with further tests. It lacks the exact same issues, as it only uses a reliability guard for execution and the fundamental data for testing is similar [44].

ClassEval. ClassEval is the first dataset containing Python tests at the class-eval. It contains 100 tasks, each with a class and about 33.1 test cases per class. In contrast to previous datasets, ClassEval explicitly mentions the use of different types of dependencies. Most dependencies are to different parts of the class itself, but some are to external "Library Dependencies". Of the 100 tasks, only 49% use imports, of which 13% use 6 different external dependencies. Even though the dataset contains external dependencies, it does not specify the exact version of the dependencies used, or the version of the Python interpreter. Also, the tests only test the functional correctness [45].

A benefit of ClassEval is the fine-grained separation of the provided code snippets. Import statement, code skeletons, class information, test cases and solution are all provided separately and can easily be combined for testing.

Conclusion. None of the datasets analyzed above fit the requirements of this thesis. Almost no datasets contain external dependencies, while none specify the exact environment. In addition, the only security measue in place is a rudimentary reliability guard, which is not enough to secure the execution of arbitrary code snippets. There is a need for a new dataset that fits the above requirements.

DependencyEval

To fulfill the above requirements, the dataset DependencyEval is proposed. It contains fine-grained information about the code snippets, including the exact environment, the used dependencies, the import statements, the code skeleton, the test cases and the solution. In addition, it contains a secure evaluation pipeline that can run the tests and collect the results reliably. This section will first introduce the structure of each entry in the dataset, followed by how the data was obtained and lastly the design of the evaluation pipeline. The samples are handcrafted to test changes in specific Python dependencies.

Structure. Each entry in the dataset shall have the following format described in Listing 52. As is usual, each entry contains a unique identifier task_id. Each entry represents a single task, targeting a single external dependency. The task_name is a name of the task containing the name of the targeted dependency and the index of the task for that dependency, 1-indexed, such as pytorch_1.

The following fields are combined to form the full solution:

```
{import_statements}

{context}

{function_signature}
{function_documentation}
{solution}
```

Listing 14: Combination of fields to form the full solution in DependencyEval

The import_statements contains a list of import statements to be performed at the start of the code snippet. The context field contains additional definitions of classes, functions or variables that are required for the code snippet to run. The remaining three fields form the function body. By combining the fields without the solution, the code snippet can be used as input to a neural code completion model, to compare against the gold standard solution. Each docstring is a Python docstring in the Google docstring format [46], which was chosen over the many other docstring formats, such as ReST [47], as it is widely used and has a compact layout.

To reproduce the exact environment, the package_dependencies field contains a list of all external dependencies used in the code snippet, pinned to a specific version. As this dataset is geared towards Python, the python_version field contains the version of the Python interpreter used to run the code snippet. This prevents any issues with dependencies needing a specific version of Python and else failing.

The field test_code contains the full test cases for the functional and approach correctness. It leverages the builtin unittest module for defining the test cases [48]. To run the full unit tests, the generated above code is prepended to the test_code field.

Further fields are for transparency and interpretation of the results. They contain additional information on the name of the function to be executed (entry_point), why, when and where the code snippet was sourced. The date field contains the date the dependency was modified, enabling e.g. quick analysis of if the code snippet could be contained in a training dataset. changelog is a direct link to the changelog of the dependency, detailing the change made. The "why" is stored in the reason field, detailing why the dependency and feature was used. This is one of:

- modification: The dependency was recently changed.
- uncommon: The dependency is rarely used and most likely unknown to a model.

The kind field contains the code kind targeted by the task, such as a changed field, parameter, function, method or block. The exact type of modification is stored in the modification_kind field, such as addition, removal, deprecation or rename, where rename encompasses all types of changes in which the functionality has moved to a different section of the dependency.

Data sourcing. The data is obtained by searching for public Python dependencies on the Python Package Index (PyPI) [49] that have a changelog. Repositories are then selected based on the different mentioned scenarios. Additionally, only the repositories are kept, for which test cases can easily be written and executed, removing dependencies requiring a complex setup to use the changed functionality.

Currently, the dataset contains 25 entries for 16 different dependencies:

Table 5: Dependencies in DependencyEval

Name	Stars	Uses in tasks
bidict	1437	2
dateutil	2282	1
dotted	181	2
emoji	1852	1
fastapi	73298	1
numpy	27356	1
pandas	43018	1
polars	27881	1

Name	Stars	Uses in tasks
pydantic	19643	3
pytorch	80214	3
rich	48161	2
sklearn	58838	2
sqlalchemy	9145	1
textual	24102	2
theflow	1	1
tsv2py	0	1

The following two examples serve as important test cases for the evaluation of neural code completion:

Python

```python
from typing import Any, Dict

from pydantic import BaseModel


class User(BaseModel):
    name: str
    email: str
    age: int

def convert_user_to_dict(user: User) -> Dict[str, Any]:
    """Convert the user into a Python dictionary.

    Args:
        user (User): Pydantic user model

    Returns:
        Dict[str, Any]: User attributes as a Python key value mapping
    """
    return user.model_dump()
```

Listing 15: Example code snippet for Pydantic showing the correct completion

Listing 15 targets the major change from **Pydantic** version 1.x.x to 2.x.x, renaming a few core methods [50]. **Pydantic** is a data validation library for Python, capable of converting complex data types to and from Python dictionaries. To convert a **Pydantic** model to a dictionary, previously the method **dict** would be called. This was renamed to **model_dump**. The previous method is still available and marked as deprecated. As the library is widely used and the change in April 2023, lots of publicly available code snippets contain the old method. In addition, the standard library method **dict** is used similarly, making it hard to distinguish between the two.

```python
from textual.widgets import TextArea


def create_textual_text_area_with_indent() -> TextArea:
    """Create a TextArea widget, which indents its content when tab is pressed.

    Returns:
        TextArea: New instance of TextArea with indentation on tab press
    """
    return TextArea(tab_behavior="indent")
```

Listing 16: Example code snippet for Textual showing the correct completion

While the previous example targets the dereferencing of a method, the example Listing 16 targets the signature of a function. The parameter tab_behaviour was renamed to tab_behavior in the widely used library Textual in February 2024 [51]. As the change occurred recently, we assume and later evaluate that most CLMs were trained on older versions and do not complete the correct parameter.

Both examples serve as core test cases for the different included information sources, such as completion items and signature help.

Evaluation pipeline.

The evaluation pipeline takes a version of the dataset, a list of models and a list of hyperparameters for the models as input. For each different combination, the pipeline runs the approach described in Section 3.4 on the code snippets in the format described in Listing 14 (without the solution field). Additionally, it runs a neural code completion without the approach, to serve as a baseline. Before running the code completion, the pipeline must first create a virtual Python environment (venv) in which all of the dependencies for the task are installed in. The code completion occurs inside the venv, such that the approach has access to dependency information.

Afterward, the generated code snippets are evaluated inside Docker containers and the results transmitted back to the pipeline via the standard output. Docker is a containerization platform that allows for the isolation of processes and the restriction of resources. Through the use of Docker, the pipeline can restrict access to resources, such as the number of CPUs [52].

To ensure that the pipeline is robust, all calls to the containerized evaluation or neural code completion are wrapped in a timeout and errors are caught and logged.

4 Implementation

After having analyzed techniques and previous work based on the requirements, Chapter 3 showed the need for an IBMS and custom dataset. The IBMS is used in the proposed approach to inject context into the prompt "during" generation, while the custom dataset can be used to obtain results and evaluate these.

In contrast, this chapter implements the proposed approach and dataset. The implementation of the proposed approach includes both the use of the IBMS and further implementations for information filtering and logits manipulation. This section will use the components in Figure 4 and only mention if the behavior is part of context injection and thus IBMS.

4.1 Language Server Protocol aided generation

Next to detailing the implementation of the various components, this section will shortly introduce the general project setup, including the used dependencies, the structure and other hints. By transparently providing details on the whole setup, the reproducibility shall be promoted.

Project setup
The general implementation of the proposed approach is versioned through Git and can be found here: https://github.com/data-niklas/llm-lsp. The project contains the installable Python package llm_lsp, which enables the usage of the proposed approach in any Python project, such as an evaluation pipeline.

To further simplify the installation, this repository includes a Dockerfile, which can be used to build a Docker image containing all necessary dependencies, to run the approach on GPUs.

Dependencies
This section will list the main package dependencies of the project and their respective uses. The two main dependencies are Transformers (transformers) and PyTorch (torch), as introduced in Section 2.4 and found in the appendix.

Transformers enable neural code completion and provide high- and low-level control over the model's behavior. The implementation will integrate with this library to provide seamless LSP-aided code completion with any supported code completion model. To manipulate tensors and perform computations, the implementation uses PyTorch. Communication to a language server is done through pygls, as it enables asynchronous communication between language servers and clients. The types of the LSP specification are implemented by lsprotocol. The different asynchronous packages rely on asyncio for asynchronous programming.

Lastly, the used language server for the Python language is python-language-server, an official language server for Python maintained by the community. It can be installed as a Python package and is developed in Python itself.

Structure
The structure of the llm_lsp package is as follows:

```
› tree -L 1 --dirsfirst
├── code_utils
├── debug
├── generation_utils
├── interrupts
├── lsp
├── mixins
├── prompt_formatters
├── config.py
├── constants.py
├── generator.py
├── __init__.py
├── __main__.py
└── prompt_state.py
```

Listing 17: The directory structure of the llm_lsp package.

The main component Generator is implemented in generator.py. Different feature sets of the generator, such as logging, editing token sequences and padding the edited sequences are extracted into the directory mixins. The different Monitors LSP Processor, Boundary Processor and Comments Processor are contained in the directory lsp, as well as further utilities needed to interact with a language server. The component Prompt State is implemented in prompt_state.py, while the Prompt Formatter is implemented in the prompt_formatters submodule. Lastly, the submodule interrupts contains the behavior for the different types of interrupts.

The other files and submodules contain various types of utilities, fixes, and configurations needed to enable the approach and make it configurable for evaluation. The next sections will focus on the components introduced in Section 3.4 and then continue on with important details on the various other files and submodules.

Environment Separation

The approach separates the runtime environment of the approach from the project environment, to prevent conflicts in dependencies. While e.g. the approach requires a specific version of the Transformers library, the project may also use Transformers, but a different version. The runtime requires one version for code completion, while the other version is required for the language features, leading to a conflict.

The approach is started while in the runtime environment. The language server uses the project environment, and some functions are run in the project environment. This will be detailed in later sections.

Configuration

All behavior of the approach can be toggled through fields of the data class LspGenerationConfig. This enables the analysis of certain behaviors on the performance of the approach, as well as introspection through logging.

The LspGenerationConfig is defined as in Listing 18:

```python
# llm_lsp/config.py
@dataclass
class LspGenerationConfig:
    boundary_processor: bool = True
    comments_processor: bool = True
    lsp_processor: bool = True
    enabled: bool = True
    chat_history_log_file: Optional[str] = None
    force_custom_pad: bool = False
    predict_correct_completion_symbol: bool = True
    masked_gen: bool = True
    use_completion_context: bool = False
    use_deprecation_context: bool = True
    use_signature_context: bool = True
```

Listing 18: The LspGenerationConfig data class, enabling customizable behavior.

The first fields enable the toggling of the Boundary Processor, Comments Processor and LSP Processor, which were introduced in Section 3.4 and are monitors. To quickly disable the whole approach, the enabled field can be set to False to overwrite any previous value.

To enable logging, the chat_history_log_file field can be set to a file path. The logging behavior is introduced in the Introspection section.

During the initialization of the tokenizer and model, the Generator may add a custom padding token, if no padding token is present. Setting force_custom_pad to True can force the addition of the padding token.

The predict_correct_completion_symbol field toggles custom behavior for the completion item context injection. If enabled, the list of retrieved completions is presented to the model, to directly predict the correct completion item. Then the predicted completion item is directly inserted into the prompt, before resuming the generation.

The next toggle masked_gen controls if the logits of "other" token ids are masked during generation, when completion items or signature help is available.

The final three toggles use_completion_context, use_deprecation_context and use_signature_context control the behavior of the LSP Processor. If disabled, the respective context is not injected into the prompt, as the monitor will not cause the respective interrupts.

The implementation additionally extends the Huggingface generation config, adding support for the field max_interrupts. If specified, the generation will terminate after at most max_interrupts interruptions.

Enabling the beam search strategy

As discussed in Section 3.3, the approach must support generation strategies like the beam search. This requires the following:

1. Storing the final sequences before selecting the single returned sequences.
2. Storing the sequence selections during generation to reproduce reorderings on objects tracking state for each sequence.

The classes providing support are the BeamTracker and BeamIndexStoringSearchScores, found in generation_utils/beam_tracking.py. The BeamTracker class only tracks sequence selections. It contains a list of tuples, where each tuple contains an index for each sequence. It forms a $|\text{sequences}| \times s$ matrix T, where s is the number of steps and is increasing during the generation loop. $T_{j,i}$ denotes that during step i the sequence j was selected from the $T_{j,i}$ sequence at step $i-1$.

By receiving a list of objects storing state for each sequence, the BeamTracker can replay the selection process for all steps, reordering the objects. As several sequences may be selected from the same previous index, the objects may be duplicated.

```python
def rearrange_according_to_beams(self, items):
    indices = self.get_final_beam_indices()
    if indices is None:
        indices = list(range(len(items)))
    return [items[indices[i]] for i in range(len(items))]
```

Listing 19: The rearrange_according_to_beams method of the BeamTracker class, which reorders the items according to the sequence selections.

The second piece of the puzzle is the BeamIndexStoringSearchScores class. It is a subclass of the Transformers BeamSearchScorer class. The BeamSearchScorer classes are used internally for the sequence search strategy to process a single generation step to choose the next sequences and to finalize the selection at the end of the generation process. By subclassing the BeamSearchScorer class, the BeamIndexStoringSearchScores class has access to the indices of selected sequences at each step. The instance contains an instance of the BeamTracker class, tracking the indices at each step. Additionally the BeamIndexStoringSearchScores class is called with the final sequences before the final selection of the singular best sequence directly before the generation loop ends. These input_ids are stored internally for external access by the Generator class. Concretely, the class overwrites the two methods process and finalize, adding the additional behavior for each step and for the final step, then passing all arguments into the method on the superclass:

```python
# llm_lsp/generation_utils/beam_tracking.py
class BeamIndexStoringSearchScores(BeamSearchScorer):
    def process(self, beam_indices, ...):
        self.beam_tracker.track_beam_indices(beam_indices)
        return super().process(...)

    def finalize(self, input_ids, beam_indices, ...):
        self.input_ids = input_ids
        self.beam_tracker.track_beam_indices(beam_indices)
        return super().finalize(...)
```

Listing 20: The BeamIndexStoringSearchScores class, which tracks the indices of selected sequences at each step and stores the final sequences before the final selection of the singular best sequence.

A later section will detail how the BeamIndexStoringSearchScores replaces another instance of the BeamSearchScorer.

StoppingCriteria

Three custom stopping criteria are implemented for the approach. The first class, InterruptStoppingCriteria, enables the core interrupt mechanism. The last two criteria are enhancements, reducing the generation time for verbose models.

The InterruptStoppingCriteria class is a utility class, which is used to determine if an interrupt occurred inside a monitor and the generation loop should be stopped. Colloquially, it propagates the interrupt signal to the interrupt loop. It subclasses the Transformers StoppingCriteria class. The class implements the method seen in Listing 21:

```python
# llm_lsp/interrupts/__init__.py
class InterruptStoppingCriteria(StoppingCriteria):
    def __init__(self, interrupt_token_id):
        self.interrupt_token_id = interrupt_token_id

    def __call__(self, input_ids, scores, **kwargs):
        # Any batch ends with the interrupt token
        for i in range(input_ids.shape[0]):
            if len(input_ids[i]) > 0 and input_ids[i][-1] == self.interrupt_token_id:
                return True
        return False
```

Listing 21: The InterruptStoppingCriteria class, which determines that the generation loop should be stopped when the interrupt token id occurs.

The next two classes CodeBlockEndStoppingCriteria and CodeBlockEndStoppingCriteria are both enhancements and serve the same purpose. Each class stops the generation when a finishing structure is detected in the last n tokens. The first class is used in the case of the regular code generation, where the generation is stopped, when a closing markdown code block is detected in the last 10 tokens. The second class is used when the predict_correct_completion_symbol configuration is enabled. It stops the generation of the predicted code completion symbol. The generation is stopped when encountering a ‘’ in the last token, as the model output should be enclosed by the character.

For each class, the token ids of the last n tokens are detokenized at each step, after which a simple string search is performed.

This drastically reduces the generation time in cases, where the model follows the desired output by several sentences describing the response, even when discouraged by the prompt.

Generator

The Generator component is implemented by the class Generator in generator.py. It is the main component of the approach and orchestrates the generation process. The idea is to provide a high-level interface to the user, somewhere between the interface of the generate function of CLMs and the pipeline abstraction in the Transformers library. The interface shall provide the same interface as the generate function, while reducing a few pitfalls through the features of the pipeline abstraction, and the added configuration options for the approach.

The following sections include notes on the different features provided by the Generator.

Initialization. The Generator is the main entry point into the approach and is initialized

using a CLM (Transformers GenerationMixin), the fitting tokenizer and a generation config. Next to the parameters essential for a default generation, it additionally receives the list of enabled interrupt types a LspGenerationConfig object. By default, the three interrupt types completion, deprecation and signature are enabled. Configuring the behavior of the approach through the LspGenerationConfig object is detailed in a later section.

Then the embeddings of the passed model and the vocabulary of tokenizer are initialized, by adding the special tokens needed for the IBMS. Additional special tokens are added to the tokenizer through the add_special_tokens method. Then the model embeddings are resized to match the new vocabulary size through the resize_token_embeddings method.

Not every model has a padding token[1]. As the approach relies on padding tokens to pad the token sequences, the initialization adds a custom padding token to the additional special tokens, if no padding token is present. In addition to adding the new padding token, the custom padding must be specified as the used padding token through the pad_token_id attribute of the tokenizer. The padding token id can be obtained from the tokenizer through the convert_tokens_to_ids method after adding the new token.

The Generator performs two distinct types of explicit initialization: the general initialization and generation specific initialization. The general initialization creates objects upfront, which can be reused between neural code completions, as they do not depend on the input code. In contrast, the generation specific initialization is performed for each neural code completion and is dependent on the input code. The generation specific initialization will be detailed in the later section detailing the interrupt loop.

The general initialization creates both a beam tracker (BeamTracker class) and an instance of a PromptFormatter class. The BeamTracker is the solution to enable support for the beam search generation strategy, as introduced in Section 3.3 in Generation Strategy Support. It will be further detailed later in conjunction with the tweaked generation loop. The PromptFormatter is a class directly implementing the Prompt Formatter component. The initialization picks one of two classes: An instance of VanillaPromptFormatter is chosen if no context is to be injected into the model, while an instance of DefaultPromptFormatter is chosen for the normal preceding of the approach. The toggle allows for a seamless switch between a baseline and the approach itself, through the passed configuration.

Tensor initialization. Another form of implicit initialization is the control of the default device placement. The device placement regulates on which physical device PyTorch tensors are placed on initialization. It is important that all tensors used during the approach are placed on the same device, else the generation fails.

To control the device placement, the Generator class implements a context manager [54], which changes the default device to the device the model was initialized on. The context manager is implemented through the method device_placement in the mixin PipelineMixin. It sets default device through the torch.device function and is inspired by the same behavior occurring in the high level abstraction pipeline in the Transformers library.

[1]CodeLlama does not define a padding token [53].

Before starting the neural code completion, the context manager is entered to ensure that all tensors are placed on the correct device. The context manager is exited after the neural code completion is finished:

```python
# llm_lsp/generator.py
async def completion(...):
    with self.device_placement():
        return await self._completion(...)
```

Listing 22: Entering the PyTorch device context manager before passing all arguments to the wrapped internal completion method.

Completion. This section details the interruption loop, which is the core of the enhanced neural code completion. As noted in the previous section, the actual completion is implemented in the internal method _complete, as the method complete wraps the method, placing all initialized tensors on the same device as the models tensors.

A shortened version of the completion loop is displayed in Listing 23:

```python
# llm_lsp/generator.py
async def _complete(self, code: str, repo_root: str, file_name: str) -> str:
        self.log_code(code, "START")
        (...) = self.initialize_generation_state(code, repo_root, file_name)

        prompt = prompt_states[0].format(code)
        input_ids = self.tokenizer(
            prompt, return_tensors="pt", add_special_tokens=False
        ).input_ids
        input_ids, _ = self.expand_input_ids(input_ids, config)
        while True:
            decoded_text = self.resume_generation(input_ids, ...)
            interrupt = lsp_processor.interrupt
            if interrupt is None:
                prompt_state = prompt_states[0]
                return self.retrieve_final_code(prompt_state, decoded_text)
            stopped_beam_index = self.index_of_beam_to_edit(interrupt.input_ids)
            prompt_state = prompt_states[stopped_beam_index]
            if max_interrupts is not None:
                if max_interrupts == 0:
                    return self.retrieve_final_code(prompt_state, decoded_text)
                max_interrupts = max_interrupts - 1
            code_util = code_utils[stopped_beam_index]
            edited_prompt = self.edit_generation_text_for_completion(
                decoded_text, prompt_state, interrupt, code_util
            )
            input_ids = self.edit_input_ids(
                interrupt, edited_prompt, stopped_beam_index
            )
            self.log_interruption(input_ids, interrupt_count)
```

Listing 23: The interrupt loop, which is the core of the Generator class and which enables the IBMS. Long argument lists are abbreviated by "..."

The core of the loop consists of Line 11 - Line 16 and Line 27 - Line 29. The first lines call the resume_generation method, which wraps around the modified Transformers

generation loop. Lines 12 to 15 check if an interrupt occurred and return the result if the generation ended with an EOS. Else line 26 edits the changed prompt into the current input token ids for the next call to resume_generation.

To further enable introspection, the initial code is logged and the input token ids are logged between interrupts. The logging behavior is detailed further in the introspection section.

At the start of the method, the generation specific objects are initialized, such as the different monitors, the prompt state, code utilities and other objects. While the monitors work on the input token ids as a whole, other objects work on singular sequences and are initialized for every sequences separately. Thus, the PromptState, PromptFormatter and CodeUtil instances are created as a list for every single sequence.

Additionally, the initial input token ids are obtained by creating the initial prompt from the provided code and then tokenizing the result. Usually the prompt is tied to a sequence and formatted using the PromptFormatter instance associated with a sequence. At the start of the neural code completion all sequences are the same, which is why the first PromptState instance is used.

Other parts of the method handle the augmented generation config. Initially, fields supported by the approach, but unsupported by Transformers are removed from the config dictionary. Then during each iteration, values such as the maximum allowed time (max_time) are updated to reflect the then correct values. The supported maximum interruptions' configuration value is tracked by the maximum_interrupts variable.

Line 10 is further explained in a later section. The same is true for details on editing and resumption.

Resumption. Although resumption refers to the continuation of the generation process after an interrupt, there is no functional difference between the first generation and generations following an interrupt. In each case, the interrupt loop calls the resume_generation method. It is a wrapper over the tweaked generation loop, which is detailed in the next section.

Firstly, it resets objects used in the previous generation. Secondly, configuration values are set for the internal generation loop. At last, it processes the result of the generation loop, to enable support for different strategies and to handle the special interrupt case.

In particular, the wrapper resets both the LspProcessor and the BeamTracker instances, both clearing any previous interrupts and resetting the tracked selected sequences.

Depending on the passed LspGenerationConfig, the different monitors are appended to a list of used LogitsProcessor passed to the generation loop. Additionally, an InterruptStoppingCriteria instance is created and added to the parameters of the internal wrapped generate method.

The resume_generation method passes the parameters return_dict_in_generate=True and output_scores=True, influencing the structure of the returned result of the wrapped generation loop. The result is a dictionary-like object containing the returned final sequences over all batches, of which there will be a single one in most cases. This returned sequence

is then processed and returned by the resume_generation method. First, any type of custom padding is removed, then any interrupt token is removed if found. At last the sequence is decoded into a text containing only the prompt without other special tokens.

Further steps are necessary to support the beam search strategy. As will be discussed in the LspProcessor section, it stores the final input_ids of the generation at the time an interrupt is set. These input_ids can be used to inject the context and then to resume the generation. If the beam search strategy is used, these input_ids could have been reordered after the interrupt token has been set. Thus the input_ids need to be overwritten with the correct input_ids after the beam search strategy has applied its own logic. In that case the correct input_ids are obtained from the generation result, which contains the value in the field beam_input_ids. This is not the default behavior and was added in the patched generation loop and will be discussed in the next section.

Furthermore, the list of PromptState and CodeUtil are reordered to match the new sequence selection, using the BeamTracker.

Tweaked Model Generation. The internal generation loop is implemented in the resume method located in the InterruptMixin class. The resume method is a patched copy generate function of the Transformers GenerationMixin. The following patches were applied:

1. No input_id expansion is applied.
2. Beam search strategy generation loop calls are tweaked to use the BeamIndexStoringSearchScores class and return the final beam search input_ids.

Usually the generate function receives input_ids, which are expanded to fit the number of sequences required by the decoding strategy. At the end of the strategy these sequences are then reduced to a single result. For an IBMS to work, the method must receive all sequences upfront and return all sequences at the end. Thus, the expansion of the input_ids is extracted into the additional method expand_input_ids. It will be called at the start of the **interrupt loop**, instead of the start of the generation loop.

Secondly, every beam search strategy must use an instance of BeamIndexStoringSearchScores as discussed before. Instances of BeamSearchScorer are replaced with BeamIndexStoringSearchScores, additionally forwarding the instance of the BeamTracker into the instance, enabling the tracking of the sequence selection process. After the generation loop has finished for a beam search strategy, the stored input_ids are extracted from the instance and set in the generation result in the beam_input_ids field.

Editing. This section details how the prompt and then input_ids are edited to inject the new context. The behavior is located in the TokenSequenceEditMixin class and is called in Line 24 and Line 27.

The first method edit_generation_text_for_completion is called with the decoded text and the context to inject. The decoded text is the prompt of the interrupted sequence, as that interrupt token led to the selection of the sequence as the result. The context to inject was retrieved from the instance of the LspProcessor. The method performs one of the following two steps:

1. Inject the context into the PromptState.
2. If the option predict_correct_completion_symbol is enabled and a completion is present: Determine the next symbol from the given completions and insert it directly into the prompt.

In any case, the methods starts by retrieving the generated code from the prompt, using the PromptState instance. And finally, the generated code is formatted into the edited prompt using a maybe changed PromptState.

In the first case, the interrupt is used to format the retrieved context into a textual representation, insertable into the prompt, called "comment". The exact steps are discussed in the later section Interrupt Types. The comment is then added into the PromptState with the interrupt type.

The second case is an experimental method to try and find the correct completion from the list of completion items directly, without inserting the list of completions and implicitly letting the model complete the item. The method determine_next_symbol_from_completions is called, which starts a separate normal model generation. The model is provided the following:

1. The current completed code.
2. The list of non deprecated completions.
3. A comment created from the deprecated completions.

While the first two points are expected, the third point is added to counteract biased completions by the model towards deprecated items. The format of the prompt can be seen in Listing 24:

The following symbols are code completion entries. Determine the appropriate symbol to complete the code in the code block: `completion_1`, `completion_2`, ..., `completion_n`

```python
def input_task(obj):
    """Docstring"""
    return obj.
```

Return only the single chosen symbol. Do not provide commentary. Output format:
`CHOSEN SYMBOL`

Hint: method a is deprecated. Use method b instead.

Listing 24: The format of the prompt used to predict the next symbol from the list of non deprecated completions directly.

The output is expected to be the completion item wrapped in backticks, as usual for raw Markdown text. If the model did not return a result in the expected format, the empty string is returned. Else the predicted completion is returned. The predicted completion is then directly appended to the generated code.

Introspection For any type of neural model generation, introspection into the generation flow is crucial. The result determined by the model can be often times unexpected and confusing. For complex approaches further manipulating the generation flow, introspection is even more so important, as the complexity hinders the understanding of the generation flow.

To enable understanding on how additional context affects the generation flow, the Generator class implements a logging mechanism in the LogMixin class. The core idea is to log the generated token sequences to an append-only log between interrupts. This enables the user to understand how previous interrupts led the model to the token sequences logged in following interrupts. Additionally, reordering of sequences can be observed.

At its core, the LogMixin provides two methods: log_interruption and log_code. The method log_code is used to log initial and final code states of singular provided and returned code. Between interrupts the second method log_interruption is used to list the contents of each sequence. For readability, the token id sequences are converted back to tokens through the tokenizer.batch_decode method.

Generally, the methods have their structure displayed in Listing 25:

```python
# llm_lsp/mixins/log_mixin.py
def log(self, text: str):
    with open(self.config.chat_history_log_file, "a") as f:
        f.write(text)
def log_interruption(self, input_ids, interrupt_index):
    if self.config.chat_history_log_file is None:
        return
    code_snippets = self.tokenizer.batch_decode(input_ids)
    # ... text formatting into variable `text` ...
    self.log(text)
```

Listing 25: The general structure employed for logging by the LogMixin.

The logging mechanism is optional and can be enabled as discussed in the configuration section. If disabled, the methods are no-ops.

Monitors

After discussing the complicated details of the interruption loop and prompt editing, this section provides details on the different monitors LSP Processor, Boundary Processor and Comments Processor. These classes receive and use state objects created in the Generator class. The following sections will only list further created objects and the state objects will not be listed explicitly.

LSP Processor Overview. All processors implement the LogitsProcessor interface of the Transformers library through the __call__ method. They receive the input_ids for each sequence and the current scores of the next predicted token id. The implementation of the __call__ method is found in Listing 26:

```python
# llm_lsp/lsp/lsp_processor.py
def __call__(self, input_ids: LongTensor, scores: FloatTensor) -> FloatTensor:
    """Returns a 2d FloatTensor which has scores for every batch"""
    if not self.config.lsp_processor or not self.config.enabled:
        return scores
    for i in range(input_ids.shape[0]):
        try:
            scores[i] = self.scores_for_batch(i, input_ids[i], scores[i])
        except InterruptGeneration as ig:
            scores[i][self.interrupt_token_id] = INTERRUPT_LOGITS_SCORE
            self.interrupt = Interrupt(
                input_ids=input_ids,
                interrupt_context=ig.context,
                interrupt_type_name=ig.interrupt_type_name,
            )
            return scores
    return scores
```

Listing 26: The __call__ method of the LSPProcessor class.

The general flow is as follows:

1. If the LSP Processor is disabled through the configuration, the unchanged scores are returned.
2. Each sequence is processed individually, after which the updated scores are returned.
3. For the current completed code in each sequence, the following occurs:
   1. Static analysis is performed on the code.
   2. Heavy filtering is applied to the retrieved completions and signature help.
   3. An interrupt may be triggered, if context is available, throwing an InterruptGeneration exception.
   4. The scores are reranked using the available context, if no interrupt is triggered.
4. If an interrupt is triggered, the interrupt token id will be forced and the context saved in an interrupt object.

To perform these actions, in special the static analysis, filtering and reranking, the monitor requires a few additional attributes.

The first additional attribute is the signature_cache, which overlays the plentiful information from the completion items with the signature help to enable complex filtering on the signature help items. By default this is initialized to an empty dictionary, which grows during the generation.

The second attribute is the interrupt, which is set to None at the start of the generation. If an interrupt is triggered, the interrupt object is set in the interrupt attribute. The method resume sets this object to None, effectively clearing the interrupt information.

The third attribute file_names contains a file name per code to complete. This name serves is used to create a temporary file containing the partial code for static analysis. It is a direct result of the file_name argument of the _complete method in the Generator and is relevant if code in a project file is to be completed.

Lastly, the expand_size attribute is used to determine the link between the number of total sequences, sequences per batch and batch count. Usually the different functions use

flat lists which contain entries for all batch entries. To determine the batch index from an index of such a flat list, the expand_size contains the number of sequences per batch.

LSP Processor Context Retrieval. This section and the following section on the LSP Processor will provide details on the behavior of the scores_for_batch method, which contains the retrieval, filtering, interrupt triggering and reranking of scores.

The method starts with the following logic seen in Listing 27:

```python
# llm_lsp/lsp/lsp_processor.py
def scores_for_batch(
    self, i, input_ids: LongTensor, scores: FloatTensor
) -> FloatTensor:
    """Returns a 1d FloatTensor for a single batch"""
    current_code = self.current_code(i, input_ids)
    file_name = self.file_name(i)
    with LspCodeFile(
        file_name, current_code, self.lsp_clients[i // self.expand_size]
    ) as lsp_code_file:
        if self.should_complete(current_code):
            completions = lsp_code_file.ask_completions()
            completions = lsp_code_file.resolve_completions(completions)
        else:
            completions = []
        trigger_phrase = re.search(r"[A-Za-z_]*$", current_code).group(0)
        signature_help = lsp_code_file.ask_signature()
        self.increase_signature_cache(signature_help, completions)
```

Listing 27: Start of the scores_for_batch method, listing how the context retrieved from a language server.

The method starts by determining the partially completed code of the current sequence using the current_code method. The current_code method reuses the PromptState abstraction to do so. The only new difficulty lies in finding the correct PromptState instance, as the LSPProcessor stores an instance per sequence and the ordering of the stored instances and the passed input_ids could be different due to beam search reordering. Generally, the class uses the BeamTracker to determine the reordered indices, then retrieve the correct object of the current sequence and then use the object to perform the action.

Similarly, the file name of the current batch is retrieved using the get_file_name_for_batch method.

Lastly, the abstraction LspCodeFile is used to interact with the language server. The LspCodeFile abstraction is detailed in the Language Server section. The returned lsp_code_file object is used to firstly retrieve the completion items and then the signature help items. As detailed in Section 2.5, retrieving the completions is a two-step process, as the completions must be resolved to include detailed information needed for filtering.

To prevent the use of completion items, if the completion is not needed, the method should_complete is called. If the current code ends in one of the following symbols, the completions are not needed: “)”, “\n”, “ “, “\t”, “]”, “}”. This effectively limits the use of completion items to the scenarios:

- completion of a symbol,
- start of the dereference operator,

and reduces completions of single function calls and unneeded symbols.

To prepare for the filtering, a `trigger_phrase` is determined. The trigger phrase is used to determine if a completion item is already partially completed, the current code ends in a prefix of a completion item. This trigger phrase is obtained by finding any symbol the code ends with through the regular expression `[A-Za-z_]*$`.

Lastly, the signature cache is increased using the retrieved completion items. The method `increase_signature_cache` extracts the keyword from builtin completion items. If no item is stored in the cache under that keyword, the completion item is assigned to the keyword in the `builtin_cache` attribute.

LSP Processor Filtering. First the signature help items are filtered, after which the completion items are filtered. Each filtering is performed through a chain of methods, each taking at least a list of items and returning the filtered list.

On the signature help items, the following filters are applied:

1. Remove signature help items of builtin functions.
2. Remove items without parameters.

The method should improve completion of functions with changed parameters and complex signatures. As such, the second filter ignores the case in which a function does not have parameters. By filtering cases where context is not needed (as the models can complete builtin functions correctly in most cases), further errors due to model controlling are prevented. To obtain the information if a signature help item is a builtin function, the signature cache is used, as the information is stored in the completion items. If a matching completion items is a builtin item, the signature help item is skipped.

From the completion items, the following filters are applied:

1. Remove builtin completion items.
2. Constrain completion items to relevant kinds.
3. Remove items from the current partial completion.
4. Constrain to public items.
5. Remove items which are already fully completed.

The steps are as described in Section 3.4. Most filtering steps are straightforward, as they only require fields directly present in the completion items, or simple string matching. The following minor details are required:

- Builtins are recognized by matching the value `"builtins"` in the `detail` field.
- Private methods are recognized through the name starting with an underscore.

LSP Processor Interrupt Triggering. As described in the design section, the filtered completion items are split into deprecated and non-deprecated items. After splitting, the code

checks if completion items, deprecation items and signature help items are to be inserted (in that specific order).

The LSP defines that completion items include a list of associated tags, which include the information if the item is deprecated. In theory this enables easy splitting of completion items into deprecated and non-deprecated items. In practice not every language server correctly detects deprecated items. The widely used language server for Python, **python-lsp-server**, does not correctly detect deprecated items.

Thus, this work implements the custom function **is_deprecated**, to properly detect deprecated items. The function relies on the behavior described in Python Enhancement Proposal (PEP) 702 [55] and reads the meta-attribute **__deprecated__** of the item, which is present in deprecated items. The function is implemented in Listing 28:

```python
# llm_lsp/interrupts/venv_language_features.py
def is_deprecated(item):
    module_name, variable_parts = module_name_and_variable_parts(item)
    try:
        module = importlib.import_module(module_name)
    except ModuleNotFoundError:
        return False
    except ValueError:
        # empty module name
        return False
    variable = module
    for variable_part in variable_parts:
        if not hasattr(variable, variable_part):
            return False
        variable = getattr(variable, variable_part)
    return hasattr(variable, "__deprecated__")
```

Listing 28: The **is_deprecated** function, which determines if a completion item is deprecated. This function is called from a subprocess using the project environment.

Listing 28 loads the module a value is stored in, retrieves the value from the module and then checks for the meta-attribute **__deprecated__**. If the field is present, the value is deprecated, in any other case it is not.

To use the function in the **LSP Processor**, the import path of the value is constructed, by concatenating the module path from the **detail** field with the name of the symbol in the following format: **is_deprecated(completion.detail + "." + completion.insert_text)**.

As this function needs to be run in the project environment, the function is extracted into a separate file **venv_language_features.py**, which is then executed using the Python interpreter of the project environment. All variables are communicated via standard streams. As importing, starting many processes is slow, and the function is called repeatedly, the results are memoized using the **lru_cache** decorator.

After splitting the items, each of the three interrupt types completion, deprecation and signature help checks if context needs to be inserted. Each of the three checks works similarly:

1. If the interrupt type is disabled in the configuration, the check is skipped.
2. If no items were found, no interrupt is triggered.
3. If items were found, they are compared to the already inserted context.
   1. If no context of the interrupt type is present, the interrupt is triggered.
   2. If context of the interrupt type is present, but different from the items, the interrupt is triggered.
   3. Else, no interrupt is triggered.

To check for inserted context, the **LSP Processor** uses the **PromptState** instance of the current sequence. To account for reordering of sequences, the **BeamTracker** instance is used to retrieve the correct instance of the **PromptState** instance.

The interrupt is thrown through the **trigger_interrupt** method, which raises the custom **InterruptGeneration** exception. It includes the interrupt type and associated context. The outer loop of the **LSP Processor** then catches the exception, sets the interrupt token id in the scores and sets the interrupt object in the **interrupt** attribute. The interrupt object has the structure seen in Listing 29:

```python
# llm_lsp/interrupts/__init__.py
@dataclass
class Interrupt:
    # Tensor of (return_count * beams * batched_items_count) x currently_generated_tokens
    input_ids: Tensor
    interrupt_context: Any
    interrupt_token_id: int
```

Listing 29: The **Interrupt** dataclass, which is used to store the interrupt context and the interrupt token id.

The object additionally contains the current **input_ids** at time of interrupt, which will be used to inject the context and for resumption. The thrown exception is strictly not needed, as the method **scores_for_batch** could have returned a tuple of the scores and maybe set interrupt. As the interrupt itself is passed through the generation through the custom interrupt token, this approach is compatible with other runtimes which do not support these types of exceptions.

LSP Processor Reranking. As described in the design section, token ids which are prefixes of a non-deprecated completion item should be ranked higher than token ids that are prefixes of deprecated completion items. Other token ids should be masked. Token ids which are prefixes of tokenized non-deprecated completions will henceforth be called **non-deprecated token ids**, and similarly **deprecated token ids** for deprecated completions.

For the reranking, the **LSP Processor** first applies a constant adjustment to the scores using the method **apply_constant_adjustments**. Then deprecated tokens ids are ensured to stay below non-deprecated token ids in the method **ensure_deprecated_below_non_deprecated**. Finally, masking occurs in the method **mask_other_tokens**.

apply_constant_adjustments applies a delta to the score of the token ids. A positive delta is applied to non-deprecated token ids, a negative delta to deprecated token ids. The delta is empirically fixed to 7.0.

In contrast, ensure_deprecated_below_non_deprecated determines the minimum non-deprecated token score. Then any scores of deprecated token ids above the minimum are fixed to the delta value below the minimum, to ensure a gap of at least the delta.

The masking is skipped if masking is disabled in the LspGenerationConfig. The implementation was adapted from the masked generation in Agrawal et al. [3]. For details see their logits processor[1][2].

When an interrupt should occur, the score of the interrupt token is fixed to 1000.0. This score was picked empirically, as all other observed scores were below 1000.0 and it is unlikely for scores to be higher. While the design section stated to use the highest possible value, the highest value ∞ did not work in practice, as the model would produce the following error:

```
RuntimeError: probability tensor contains either `inf`, `nan` or element < 0
```

Listing 30: The error message produced by PyTorch when the score of the interrupt token is set to the highest possible value.

PromptState

The PromptState class is contained in prompt_state.py and is a direct implementation of the PromptState component. At its core, it contains a list of inserted comments of the sequence index it is assigned to. Next to the comments, the PromptState uses the tokenizer to convert between the prompt and token ids. Additionally it requires a PromptFormatter instance to format the prompt into the appropiate format.

The PromptState is initialized with the initial given code and no comments. Then the private method _create_completion_prompt creates the initial prompt to the model through the following steps:

1. All comments are joined into a single string using newlines.
2. The PromptFormatter instance is used to create a list of messages, given the initial text and instruction text.
3. The messages are formatted into a prompt, using the tokenizer method apply_chat_template, into the model specific prompt format.

The method is called whenever the list of comments is changed to update the initial prompt. The initial prompt is only used for generation initially. It is used to track the prefix of the prompt, including the comments, which is then used to determine the completed code without comments, or to create a new prompt with edited comments. The comments are changed through the add_comment method, which takes a comment of an interrupt type and adds it to the list of comments. In a previous step all comments of the same interrupt type are removed, ensuring that only the most recent instruction of an interrupt type is present. This limits the scope of the context to the next context instance. For function signature context, the context would stay included until the next

[1]https://github.com/microsoft/monitors4codegen/blob/022c65efb19cf6046d0b67960ca232ae7a351af4/src/monitors4codegen/monitor_guided_decoding/hf_gen.py#L30C15-L30C49

[2]While the design of the strategy was developed independently of Agrawal et al, the implementation was retrofitted to support masked generation.

function call is completed. If the comment is empty, no instruction is added, effectively removing the previous instruction of the interrupt type.

After adding comments, the method format takes the current completed code, wraps it into Markdown code block and appends it to the initial prompt, which contains the comments. This results in the following prompt format seen in Listing 31:

<s>[INST] <<SYS>>
Return only the completion of the given function. Follow the instructions in the code comments for additional instructions on how to complete the code. Generate readable and simple code. <</SYS>>
Complete the following Python function. Return only code.
Hint: The method dict is deprecated. Use model_dump instead. [/INST]

```python
def convert_user_to_dict(user):
    return user.
```

Listing 31: A full sample prompt, including system and user prompt, comments and initial code.

To read only the generated code or the full code sample, the methods get_generated_code and get_full_code are used. get_whole_code receives a decoded text, removes the initial prompt using simple string processing, then extracts the code from the Markdown code block. get_generated_code builds on top of get_whole_code and removes the initially provided code from the full code.

Tokenization Issues. This class heavily relies on simple string processing of the prompt, i.e. removing a known prefix from the completed prompt. This works under the assumption that tokenization is reversible, that $x = \text{detokenize}(\text{tokenize}(x))$ for all prompts x. In practice this is not true. As investigated by Imgrund [56], 31% of tokenized then detokenized tokens yield a different token. This especially affects this work when using CodeLlama in the following example: "<s>[INST]" will be tokenized then detokenized to "<s> [INST]". This affects every single generation and is a critical problem. Other issues were observed infrequently. To address this issues, the method get_generated_code first fixes the received prompt by applying several string replacements to fix the issues. It e.g. replaces "<s> " with "<s>".

Interrupt Types

The approach uses the two context sources completion items and signature help items to provide the three types of interrupts "completion", "deprecation" and "signature". Each of the interrupt types is used to inform the model about available completions, deprecated completions which it should not use and signatures of functions.

Each interrupt type is implemented through a subclass from the Abstract Base Class (ABC) InterruptType, which is located in interrupts/__init__.py. The implementation of the base class is shown in Listing 32:

```python
# llm_lsp/interrupts/__init__.py
class InterruptType(ABC):
    @abstractmethod
    def type_name(self) -> str:
        pass

    @abstractmethod
    def create_comment(self, context: Any, code_util: CodeUtil) -> Optional[Comment]:
        pass
```

Listing 32: The InterruptType abstract base class, which is used to implement the different interrupt types.

As can be seen in Listing 32, each subclass must provide a (unique) type name through the method type_name, and must be able to create a comment, given a context and a CodeUtil instance. The created comment is optional. In the case that no comment is returned, the PromptState will remove previous comments as described in the previous section.

In the case of the "completion", the created comment will have the following structure:

Hint: The following symbols are code completion entries. Use the appropriate symbol to complete the current code: , , ...,

Figure 5: The structure of the comment created by the "completion" interrupt type. The insert texts of all items are joined with ",", similar to the colored bars.

For deprecation items, the comment will have the following structure:

Hint:
Hint:
Hint:

Figure 6: The structure of the comment created by the "deprecation" interrupt type. For each item, a line with a hint is created, where the deprecation message is inserted at the position of the colored bar.

For signature help items, the comment will have the following structure:

Hint: The code item has the following signature:
The code item has the following documentation:

Provide as few arguments as possible and as many as needed. Provide arguments as positional arguments instead of as named arguments if possible. Often times optional arguments can be omitted

Figure 7: The structure of the comment created by the "signature" interrupt type. The green bar is replaced by the signature text of the item, while the pink text is replaced by the shortened documentation string.

Language Server

The final section of the implementation of the approach describes any details on interacting with a language server. The following paragraphs start with the initialization of a

language server, how the static analysis is triggered and how issues between synchronous and asynchronous code have been resolved.

Initialization. As described above, generation specific state is initialized in the initialize_generation_state method. There a language server is initialized for the programming language of the code provided. The programming language is detected by the file type. The function create_lsp_for_language in lsp/__init__.py calls the correct function for the specific language. In the case of Python, create_python_lsp is called.

The dependency pygls provides the class BaseLanguageClient, which abstracts the creation and protocol communication. By calling the method start_io("pylsp") with the binary of the language server, the binary is executed in a separate process and communication occurs via standard streams. Now different method can be called, which each provide a different feature of the LSP. As described in the protocol, the initialization flow is the following:

1. Client sends a initialize request, the server responds with its capabilities.
2. Client sends a initialized notification.
3. Client can now send further requests to configure the server or retrieve information.

Respectively, the methods initialize_async and initialized are called to fulfill the first two steps. The first method has the _async postfix, as it awaits a responds using asynchronous Python code, while the second is only a notification and does not require a response. The code passes several important initialization parameters to the language server:

root_path The path to the root directory of the project. This is important to correctly determine project specific settings, such as virtual environments with dependencies.

capabilities All used capabilities are registered, such as completion and signature help requests:

tag_support Enable support for the deprecation tag on completion items.

resolve_support Enable support to receive detailed information on completion items, enabling complex filtering.

Additionally, to the LSP messages, the environment variable VIRTUAL_ENV is ensured to be set. If it is not set yet, the virtual environment is determined by the project root path and set. The environment variable is used by the language server to determine the installed packages and to provide completion items and signature help. This allows separating the runtime environment for the approach and the project environment, ensuring that the approach does not interfere with the project environment.

Temporary Code File. The completion and signature help on partially completed code is abstracted through the LspCodeFile class in lsp/file.py. The language server operates on files stored on disk, while the generated code is stored in memory. To bridge this gap, the LspCodeFile creates a temporary unique file for each partially completed code. It then notifies the language server about the new file, and then provides the completion and signature help on the file.

To determine a unique file path, the LspCodeFile uses a given path to a file as a base, which could be the origin file of the code in completion. It then appends a unique

identifier to the path, by generating a UUID1. The temporary file is then stored next to the original file, thus allowing imports to behave normally. If the code completion occurs on a singular file, the exact directory the temporary is stored in is not important.

Then a TextDocumentItem instance is created, representing the code file for the LSP. The two methods __enter__ and __close__ are used to create and delete the file. In addition, the server is notified that the file is opened or closed through the text_document_did_open and text_document_did_close methods. By defining __enter__ and __close__, the class can be used as a context manager (i.e. with statement), ensuring correct cleanup of the temporary file.

The language features, such as completion and signature help, occur at a certain Position in the document. In the case of the approach, the position is always at the end of the document. The method char_line_of_code takes the current code and returns the line and character position of the last character in the code. It first splits the code into lines, counts the last line, then counts the characters in the last line.

The three methods ask_completions, resolve_completions and ask_signature first determine the last position, then call the language server with the respective language feature request. One peculiarity is ask_completions, as the completion response may be in one of two formats. The server either returns a list of completion items or a CompletionList object, which contains the completion list in the items field. In the latter case, the completion list is extracted from the field and returned. The method resolve_completions iterates over the completion items and calls completionItem/resolve for each item separately.

Sync vs Async. The initialization and communication with the language server is highly asynchronous, as two different processes communicate through standard streams. After sending a request, the server may respond at any time, which yields an awaitable to the client. Thus, the pygls library depends on asyncio to handle the asynchronous communication. This requires the whole code to be asynchronous, such that the outer program initializes an event loop and runs the asynchronous code. Therefore, the complete method of the Generator is asynchronous, propagating the asynchronous nature into inner methods, such as the language server initialization.

The Transformers library, however, is synchronous. All implemented subclasses of Transformers classes contain synchronous methods, such as the __call__ method of the LSPProcessor. Thus, the asynchronous nature is not progragated through the generation loop to the LSPProcessor. To use the completion and signature help features, the LSPProcessor now requires access to the outer event loop, to run the asynchronous functions. To solve this problem, the asynchronous code, nested in synchronous code, nested in asynchronous code, retrieves the current event loop and calls run_until_complete on every called asynchronous function:

```python
# llm_lsp/lsp/file.py
completions = asyncio.get_event_loop().run_until_complete(completion_awaitable)
```

Listing 33: Waiting on to be returned completions from the language server. The asynchronous code is run in the synchronous code by retrieving the current outer event loop.

By default, asyncio does not allow the use of such nested asynchronous code. The implementation uses the dependency nest_asyncio to patch the asyncio library to allow nested asynchronous code.

4.2 Dataset Implementation

This section details the implementation of DependencyEval, from the creation of the dataset to the evaluation pipeline. The structure of this section is similar to the structure of the previous section.

Project Setup
The implementation is found here: https://github.com/data-niklas/DependencyEval. It is distributed as a single self-contained Python project and versioned through Git. DependencyEval provides a Command Line Interface (CLI), which includes the following functionality:

1. Creation of the distributable dataset.
2. Completion and then evaluation of the dataset on the approach, using several model configurations.
3. Calculation of metrics and export into different format.

The functionality of the CLI is split into several modules, e.g. the calculation of different metrics lies in its own module. This enables the use of this Python package as a library, reusing functionality such as the evaluation pipeline.

Data Collection
While the previous sections detail the evaluation pipeline, this section briefly details the data collection process. The process consists of crawling, filtering, scenario extraction and scenario validation.

In the first step crawling, possible Python dependencies are searched for manually, either by searching through PyPI, by browsing the web, asking colleagues or through any other means. As this is a manual process, it is inefficient, different sources provide duplicates.

While crawling and afterward, the dependencies are filtered after different criteria. Each dependency must not have too many tasks present in the dataset yet. Currently, the highest amount of tasks per dependency is 3 (pydantic). In this case pydantic would be filtered in favor of other dependencies. Additionally, the dependency must have some form of changelog, which can be used to determine potential scenarios, such as deprecation etc. Usually the dependency has a Git repository, which includes some form of changelog file in the root directory (CHANGELOG, CHANGELOG.md, NEWS, HISTORY, etc.). Alternatively, the changelog can often be found in the GitHub releases tab. Popular dependencies also have a changelog on their website. Another criteria is how well the approach correctness and functional correctness of scenarios can be tested. Crafting scenarios for some dependencies is hard, such as game engines. In the filtering process, algorithmic libraries are preferred to such dependencies, as they are easier to test.

Then, the changelog is skimmed for potential scenarios, while the previous criteria are kept in mind. If a plausible scenario is found, a task is created, including the approach

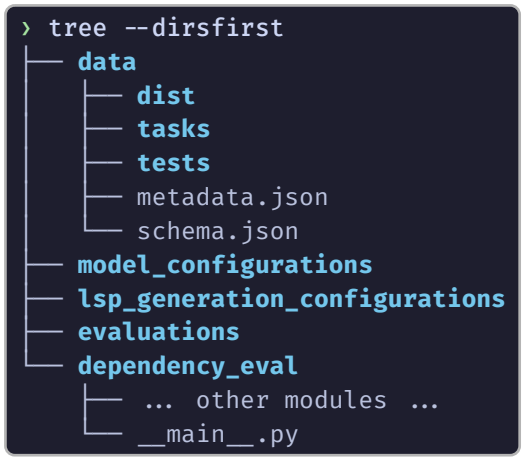
and functional correctness tests. At last, the functional tests are run to validate the gold solution.

The manual collection process is time-consuming and increases per found scenario, as it is difficult to find dependencies which match all criteria. E.g. the time spent on finding a suitable 22nd task took ~30-60 minutes, while the first three tasks took ~10-20 minutes each.

Dependencies

To achieve the listed functionality, DependencyEval relies on several dependencies: The two main dependencies are click and virtualenv-clone. click helps in creating CLI applications, as it allows to create such applications declaratively. In addition, it automatically creates a help page of the commands and subcommands. virtualenv-clone is another fundamental package in enabling reproducable and fast workflows, as it can duplicate a Python virtual environment. It is used to duplicate a virtual environment from a cache into a working directory. Other dependencies are openpyxl to export stats into Excel sheets, matplotlib to create beautiful charts and sankeyflow to create Sankey diagrams of the results without and with the approach. textual is used to display results interactively in a table, while the dependency tqdm is used to display progress bars for long evaluation runs. pygount counts lines of code and enables dependency analysis of dataset dependencies. All other features use the standard library for operations like working with files and processes.

Structure



Listing 34: Directory structure of the dependency_eval repository.

The structure of the repository is fourfold. The data directory contains all information to build the distributable dataset, which is then located in data/dist. The evaluation results of the dataset are stored in evaluations, such that they are persisted in Git. The Python package dependency_eval can be executed with Python, which runs the file __main__.py and serves the functionality through the CLI program. The evaluations use different model configurations, which are stored in model_configurations. Similar to how model are evaluated with different configurations, the approach is evaluated with different configurations, which are stored in lsp_generation_configurations.

Dataset Distribution

The distributable dataset is a .jsonl file, where each row is a JSON dictionary with the schema as introduced in Listing 52. While the compact .jsonl dataset file is easily

distributed and machine-readable, it is hard to read for a human. Thus, the basis for the dataset is an easily human-readable form, which simplifies extending the dataset by a human (which is necessary, as the dataset is curated manually). One base is the data/tasks directory, which includes a single code file for each task. The code files are named after the task and include the whole code, including the gold solution.

```python
from textual.widgets import TextArea


def create_textual_text_area_with_indent() -> TextArea:
    """Create a TextArea widget, which indents its content when tab is pressed.

    Returns:
        TextArea: New instance of TextArea with indentation on tab press
    """
    return TextArea(tab_behavior="indent")
```

Listing 35: The contents of the data/tasks/textual_1.py file, which is used to create the task textual_1 in the DependencyEval dataset. The file is split to obtain the imports, additional context, the function definition, function documentation string and gold standard completion.

As described in the design section, the fields in Listing 14 are combined to form the whole code with or without completion, depending on the use case. Using the reverse process, a file such as Listing 35 is split into the different components to form a dataset entry. This is possible, as every single file adheres to the same regular format.

For evaluation, the different unit tests are also stored in their own code files, in data/tests. To keep the tests as simple to run as possible, they do not rely on external test runners and reuse the Python unit test framework.

```python
# data/tests/textual_1.py
from importlib import reload
from unittest import TestCase, TextTestRunner, main
from unittest.mock import MagicMock


class Test(TestCase):
    def test_approach_correctness(self):
        import textual.widgets
        TextArea = reload(textual.widgets).TextArea
        globals()["TextArea"] = TextArea
        from textual.app import App
        app = App()
        out = create_textual_text_area_with_indent()
        assert isinstance(out, TextArea)
        assert hasattr(out, "tab_behavior")
        assert out.tab_behavior == "indent"

    def test_output_correctness(self):
        import textual.widgets
        TextArea = reload(textual.widgets).TextArea
        from textual.app import App
        app = App()
        TextArea = MagicMock(TextArea)
        globals()["TextArea"] = TextArea
        out = create_textual_text_area_with_indent()
        assert TextArea.call_count == 1
        kwargs = TextArea.call_args.kwargs
        assert "tab_behavior" in kwargs
        assert "tab_behaviour" not in kwargs
        assert kwargs["tab_behavior"] == "indent"

if __name__ == "__main__":
    import logging
    logging.disable(logging.CRITICAL)
    import json
    import os
    result = main(exit=False, verbosity=0, testRunner=TextTestRunner(verbosity=0,
stream=open(os.devnull,"w"))).result
    print(json.dumps([len(result.errors), len(result.failures), result.testsRun]))
```

Listing 36: The unit tests for the textual_1 task. The contents are stored in data/tests/textual_1.py. Next to the tested dependency, only the built-in unit test framework is used. The results are communicated via standard streams.

The unit tests for the textual_1 task can be found in Listing 36. The test file contains both a test for the functional correctness and approach correctness.

The final piece needed to form the distributable dataset is the additional metadata associated with each entry. While the metadata could have been embedded in the code directly, e.g. as a Python comment at the top of data/tasks/textual_1.py, a separate metadata file such as data/metadata.json simplifies the parsing process and reduces format driven errors. metadata.json does not use any special format. It contains a list for dictionaries, where each dictionary contains additional metadata in the same schema as the final dataset,

including a field task_name. The task name is the same task name used for the code file, unit test file and helps associate the different information.

When distributing the dataset, these three information sources are combined to form the different task entries. This has both the benefit of an easy-to-extend dataset for humans, while enabling easy distribution and use by machines.

Unit Tests

As introduced in the design section and quickly described in the last section, each entry requires two test cases. The first test case tests the functional correctness, while the second test tests the approach correctness.

The tests are implemented as standard Python unit tests. This simplifies the testing as the built-in unit test framework can be reused through the unittest package. The test setup requires three components to fit together:

1. Starting the unit tests from the running Python code and
2. disabling logging to access and log the test results, while
3. mocking used objects to introspect the used approach.

Typically, unit tests are either run through the command line using the unit test module, or by running the Python script directly and calling the function unittest.main.

Example 6: Typical execution of Python unit tests through the command line.

```
python -m unittest path/to/file.py
```

In either case, all defined unit tests are found and executed, after which the result is printed to the standard output. The result looks similar to Listing 37:

```
...
----------------------------------------------------------------------
Ran 2 tests in 0.200s

OK
```

Listing 37: Typical console output when executing two Python unit tests. Parsing the output is error prone and unneeded, as the results can be accessed directly.

The script accesses the test results directly through the results field, which contains the number of failures, tests with errors and the total number of tests run. The list of errors, failures, and total tests is serialized using the JSON format and sent to the controlling process through standard output. This simplifies the parsing process and easily yields the desired information and is extensible if more information is needed.

As this approach relies on standard output for Inter Process Communication (IPC), all other output must be suppressed, as it complicates the parsing of the results in the controlling process. Printing of the unit test results is suppressed by setting the test log verbosity to 0 and redirecting the output of the test runner to the /dev/null stream, effectively voiding the output.

Other output is suppressed, by setting the logging level of the logger to critical.

Each unit test is carefully crafted, by mocking the objects of the gold solution and of probable deprecated functions. Tests can then be failed directly, if the deprecated function

is used instead of the gold solution. Even if the gold solution function is used, the passed parameters can be introspected, enabling complex test scenarios. Each test scenarios test certain properties. For the scenario in Listing 36 the tests assert that the parameter tab_behavior is used, while tab_behaviour is incorrect, as it is deprecated.

Evaluation Loop

At the heart of the evaluation pipeline lies the evaluation loop, which is located in dependency_eval/loop.py. It is abstractly defined in Listing 38:

```python
# dependency_eval/loop.py
def run_loop(
    model_configurations: List[ModelConfiguration],
    dataset: Dataset,
    model_configuration_finished_cb: Callable[[ModelConfiguration], None],
    item_cb: Callable[[ModelConfiguration, Dict[str, Any]], None],
):
    for model_configuration in tqdm(model_configurations):
        tqdm.write(f"Running for {model_configuration.name}")
        for item in tqdm(dataset.items):
            item_cb(model_configuration, item)
        model_configuration_finished_cb(model_configuration)
```

Listing 38: The general evaluation loop, which combines several each model configuration with a dataset task.

The loop combines every model configuration with every dataset task. The progress is displayed using the tqdm dependency, which provides an estimate of the remaining run time, as seen in Listing 39:

```
› python dependency_eval all
 67%|████████████████████████████████          |  2/ 3 [1:44:02<34:24, 2064.64s/it]
100%|████████████████████████████████████████| 22/22 [  42:46<00:00,  116.65s/it]
```

Listing 39: Sample output of the running evaluation. The first progress bar displays the progress in evaluation whole model configurations. The lower progress bar shows the progress of a single model configuration, showing the progress in evaluation a single task.

The passed item callback is called for every different combination. By abstracting the content of the item callback, several scenarios are possible, such as only running the neural code completion, evaluation, or both. After the whole dataset for a single model configuration was processed, the model configuration finished callback is called, running any finishing task, such as saving the results for the configuration.

Robust Neural Code Completion

The completion is located in dependency_eval/generate.py. It requires a project directory with a virtual environment, in which all dependencies are installed.

The code for the passed task is built, excluding the gold solution and then written to the file code.py in the project directory. In addition, a LSPGenerationConfig object is created and serialized into JSON. Depending on if the code should be completed using the approach or not, the field enabled is toggled. Then a subprocess runs the Python binary of the project virtual environment, calling the file dependency_eval/eval_prompt.py and passing the LSPGenerationConfig, the name of the model and the path to the code.py

file as arguments. This trick enables running the neural code completion inside the exact environment as defined in the dataset. This ensures that only these dependencies are found and used, without contamination by the evaluation pipeline dependencies.[1]

Similar to before, eval_prompt.py suppressed the standard output, as it prints the completed code into the standard output.

Isolating the generation from the evaluation pipeline process increases the robustness, which is vital as the pipeline may run for several hours. If the generation should fail due to out of memory issues, issues in the communication with the language server or other places, the evaluation pipeline can resume.

Code Evaluation Sandbox

To securely evaluate foreign code, a special evaluation sandbox is created using Docker. The implementation currently only uses plain Docker. As Docker is a containerization tool, it is a vast improvement over the reliability guard of e.g. HumanEval+. But improvements can still be made by adding a tool such as https://github.com/google/gvisor, to further secure the kernel.

IPC with the evaluation sandbox again relies on sending JSON serialized messages over the standard output stream. Similar to running the code completion, the evaluation starts a subprocess, passes all arguments and handles errors. Finally, all dangling resources are removed.

The Docker container is started with the following arguments seen in Listing 40.

[1]The language server might be able to find the virtual environment, based on the passed code file alone. But the custom logic to determine if a symbol is deprecated uses importlib, reusing the dependencies available in the currently running Python instance.

```python
# dependency_eval/eval.py
def get_docker_cmd(item: Dict[str, Any], code_file: str, requirements_file: str):
    return [
        "docker",
        "run",
        "-it",
        "--rm",
        "--name",
        "dev_dataset_eval_item_" + SALT,
        "-v",
        f"{requirements_file}:/tool/requirements.txt",
        "-v",
        f"{code_file}:/code/llm_lsp_code.py",
        "-w",
        "/code",
        "--cpus",
        "1",
        # "--network", "none", # TODO: use docker build
        get_docker_image(item["python_version"]),
        "sh",
        "-c",
        "apt update >/dev/null 2>/dev/null&& apt install git -y >/dev/null 2>/dev/null && python -m venv /tool/venv && /tool/venv/bin/pip install -r /tool/requirements.txt 2>error.log >/dev/null && /tool/venv/bin/python llm_lsp_code.py || cat error.log",
    ]
```

Listing 40: Arguments to Docker run, which starts the evaluation in a Docker container. Note that Git is installed to support installing dependencies from Git repositories.

The Docker container image is an official Python image, where the Python version is set from the task. Ultimately, this creates the specific environment needed for the task. The Python package dependencies are installed inside the container, before running the evaluation. The code to evaluate is mounted into the container, next to the requirements file. The compute is limited to a single CPU.

Docker supports a filesystem size limit using the `--storage-opt` flag, which could be used to limit the size of the filesystem. In practice, this requires support by the Docker storage driver, of which filesystems like `btrfs` and `zfs` are favoured [52]. This implementation does not limit the filesystem size for compatibility reasons.

The container execution time is limited to two minutes, which is enough time to install larger dependencies, such as PyTorch. After the two minutes have elapsed, the container is forcefully removed if it still runs. To reference the container, it is given the name `dev_dataset_eval_item_`, followed by a salt. The salt is unique to the run and obtained by taking the first 8 characters of a random UUID4:

```python
# dependency_eval/constants.py
SALT = str(uuid4())[:8]
```

Listing 41: The salt is a unique identifier appended to any name used during the evaluation.

This allows for multiple evaluations to run in parallel, without interfering with each other.

Virtual Environment Management

Running the neural code completion requires a whole Python virtual environment per task, as each task has its own pinned dependencies, the project environment. As creating each environment may take up to several minutes when installing larger dependencies, the environments are cached and reused.

```
› ncdu .venv-cache
ncdu 1.15.1 ~ Use the arrow keys to navigate, press ? for help
─── .venv-cache ──────────────────────────────────────────────────────────
    4.8 GiB [##########] /cd68a527644bd64af013bf7a9f729552 ...
  324.6 MiB [          ] /0551f505a517cb4a70e339be34dbd0b9 ...
  321.1 MiB [          ] /52f3e55ccdd9f9a817225b3a93bd6f5e ...
  211.1 MiB [          ] /6e602e432e4890878c649f89a1e13d65 ...
  153.0 MiB [          ] /0d9deded4166d9f3f7cfbabad21eec78 ...
  143.7 MiB [          ] /d355ff14783721e758e4feba0296af2d ...
   88.2 MiB [          ] /d06a6a0ab58261799750c0347e607c1b ...
   83.6 MiB [          ] /34f93206053dab5422f0e18651e2ad8b ...
   83.6 MiB [          ] /5214df74f6123a5a43db9fadf18ce2a9 ...
   79.7 MiB [          ] /9949d4d6cdd06c19f58f56562aab0362 ...
   77.9 MiB [          ] /85141ce04cabb41b8818b2914542812f ...
   77.9 MiB [          ] /a8fb6cafe4e79f3fd078bf1689c4b1c8 ...
   75.6 MiB [          ] /40c1e545212e9bde75aefe582be5ae56 ...
   71.2 MiB [          ] /f8cad8f44e9edace1e261212fbaedb03 ...
   70.5 MiB [          ] /2a6c33f92b3e3c1bec67881810d952a3 ...
   67.5 MiB [          ] /71f172736178868e25791c50e7eaa923 ...
   66.9 MiB [          ] /2873f4fd489dd8d4d8cda47854b97644 ...
   66.8 MiB [          ] /7d4725890b7bc2e9d5d4e3ed68771c38 ...
   66.8 MiB [          ] /4825617287c77e0f67fbd42e53d17927 ...
Total disk usage:   3.9 GiB  Apparent size:   6.7 GiB  Items: 74870
```

Listing 42: Space used by the virtual environments in the cache. Results obtained using ncdu.
Each directory is a Python virtual environment containing specific dependencies of a task.

The dependencies are first sorted, then hashed using SHA256, to further enable reuse between tasks. The hash is used as a name for the virtual environment. If the environment does not exist yet, it is created using the venv package. Instead of the typical CLI use, the following Python functions are used, as seen in Listing 43:

```python
# dependency_eval/venv_cache.py
from venv import EnvBuilder
def create_venv(venv_directory: str, requirements: str):
    tqdm.write("Creating new venv in cache")
    builder = EnvBuilder(
        system_site_packages=False,
        clear=False,
        symlinks=False,  # Important for the Docker container!
        upgrade=False,
        with_pip=True,
        prompt=None,
        upgrade_deps=False,
    )
    builder.create(venv_directory)
    context = builder.ensure_directories(venv_directory)
    with open(REQUIREMENTS_FILE, "w") as f:
        f.write(requirements)
    cmd = [context.env_exec_cmd, "-m", "pip", "install", "packaging", "wheel"]
    subprocess.check_output(cmd, stderr=subprocess.STDOUT)
    cmd = [context.env_exec_cmd, "-m", "pip", "install", "-r", REQUIREMENTS_FILE]
    subprocess.check_output(cmd, stderr=subprocess.STDOUT)
    os.remove(REQUIREMENTS_FILE)
```

Listing 43: Creating a virtual environment programmatically in Python.

The EnvBuilder in Listing 43 creates the base environment without further dependencies. After manually installing packaging and wheel, the remaining dependencies can be installed using a regular pip install, using the Python installation of the virtual environment.

If the virtual environment exists yet, it is cloned into the code directory, such that the neural code completion can find the environment. As virtual environments contain links, copying is non-trivial and involves more than copying the directory. Thus, the dependency clonevirtualenv is used to clone the cached environment.

The environments may be prepopulated once before evaluation, by running the subcommand create_venvs.

Using the same approach the runtime environment is created once, before the evaluation starts. Only the approach is installed into the runtime environment.

Main

The main file has the following structure, as seen in Listing 44:

```python
# dependency_eval/__main.py__
import click

@click.group()
@click.version_option(VERSION)
@click.pass_context
def cli(ctx):
    ctx.obj = None


@cli.command()
@click.option("--dataset-file", default=DEFAULT_DATASET_PATH)
@click.option("--llm-lsp-directory", required=True)
@click.option("--venv-cache-directory", default=DEFAULT_VENV_CACHE_DIRECTORY)
@click.pass_obj
def create_venvs(
    args, dataset_file: str, llm_lsp_directory: str, venv_cache_directory: str
):
    dataset = load_dataset(dataset_file)
    for item in tqdm(dataset.items):
        tqdm.write(item["task_name"])
        get_venv_for_item(venv_cache_directory, None, llm_lsp_directory, item)


if __name__ == "__main__":
    cli()
```

Listing 44: Structure of the DependencyEval main file, which includes the CLI commands.

The CLI is defined through the **click** dependency and invoked by calling the **cli** function. By using the function decorator **click.group**, the general CLI is defined, with the given version VERSION. It automatically detects the package name and sets it as the program name.

Subcommands are registered, by using the **cli.command** decorator on single functions, where each function then represents a single subcommand. Single arguments are defined through the **click.option** decorator.

The **create_venvs** function defined in Listing 44 defines a subcommand with the same name, taking three arguments, of which two have a default value. All uppercase variables are constants either defined in **dependnecy_eval/constants.py**, or in other parts of the file.

5 Experimental Evaluation

After providing details on the implementation on the approach and dataset, this chapter presents the experiments conducted to evaluate the approach. After stating the environment the experiments were conducted in, the chapter presents the research questions.

5.1 Experiment Environment

The experiments were designed to use minimal resources and to be easily reproducible. They were run on a single server with the following specifications:

Table 6: The experiments were run on the server Luke, with the specifications listed in the table. Only a single GPU is required for the experiments.

Component	Value
CPU	2x AMD EPYC 7713 64-Core
Memory	1 TB
GPU	4x NVIDIA GeForce RTX 3090

The server will be referred to as Luke throughout the experiments. The code is available in the dataset repository[1].

Luke is running Ubuntu with kernel version 5.15.0 and has Docker installed. The experiments use Docker, Python 3.9.18, Transformers 4.39.0, PyTorch 2.2.1, Nvidia CUDA 11.8. All other dependencies are automatically installed through the used Docker images at runtime.

Models

As the thesis focuses on improving neural code completion, only code completion models are used. Furthermore, only instruction tuned models are used, as instruction following capabilities are crucial for context injection techniques. The goal is to improve small models, as they can easily be run locally on developers' machines and on Luke.

In particular, models of about seven to nine billion parameters are used. They offer a good trade-off between the quality of the completion, instruction following capabilities, and the computational requirements. The experiments employ models of the Llama 2 [57], Mistral [58], and DeepSeek [59] architectures. Every model is based on the Transformer architecture, each with Rotary Position Embedding (RoPE) [60] and most with Grouped-Query Attention (GQA) [61].

The first model is Code Llama Instruct 7B [62], which is based on the Llama 2 architecture and will be referred to as Code Llama. It has 7 billion parameters and supports a context length of 4096 tokens. The next model is MistralHermes-CodePro [63], a 7 billion parameter model based on the Mistral architecture, and will be referred to as MistralHermes. In contrast to other architectures, MistralHermes uses sliding window attention and was trained on a context of 8k tokens. Lastly, the Magicoder-S-DS [64] model is based on the DeepSeek architecture and has 6.7 billion parameters. It supports

[1] https://github.com/data-niklas/DependencyEval

a context size of 16k tokens and is the only model that does not have GQA enabled for the small model.

After determining that the models behave similarly, the experiments will only use the Code Llama model, to reduce the computational requirements.

Each of the previous models has support for the Transformers runtime, which enables the normal use of the models in the experiments. Next to these models, the behavior of the popular Copilot tool is of interest. While the model behind Copilot, Codex [65], cannot be integrated into the approach, it can be used for normal code completion. It will be used as a baseline.

Dataset

As shown in the design chapter, currently no dataset contains code samples of dependencies, where the whole environment is specified. For the environment, the dependency versions must be specified, and the required version of the programming language must be known. Additionally, the dataset entries must include examples of deprecated code. The design section listed several scenarios, such as the use of alternatives to deprecated code or the use of e.g. renamed function parameters.

The consequence is that this thesis created a new dataset, the DependencyEval dataset. The experiments will use the 25 hand chosen tasks from the dataset to evaluate the new approach in a controlled environment on several different scenarios. As the dataset only contains 25 tasks, a few experiment results will additionally be evaluated by hand.

Model and Approach configurations

The DependencyEval dataset is used in conjunction with the listed models, to evaluate neural code completion both with and without the approach. The evaluation using normal code completion is used as a baseline to compare the results with the approach. The three evaluations are run using the following three approach configurations:

Default: The config as seen in Listing 18 without masked generation.

Masked: The config as seen in Listing 18.

Use completion: The config as seen in Listing 18 without masked generation, but using the completion context.

Additionally, the six model configurations are used:

- Sampling with 3 beams,
- Sampling with 2 beams,
- Sampling without beam search,
- Greedy decoding with 3 beams,
- Greedy decoding with 2 beams,
- Greedy decoding without beam search.

Each model configuration includes the base parameters seen in Listing 45.

```json
{
    "num_return_sequences": 1,
    "max_new_tokens": 2048,
    "repetition_penalty": 1.3,
    "max_time": 90.0
}
```

Listing 45: Base model configuration used for all models.

The base parameters ensure that the model finishes in at most 90 seconds, which is especially important as to limit the total evaluation time. The repetition penalty of 1.3 has proven itself to prevent repetition of e.g. \\n in CodeLlama, over a value of 1.1. Additionally, sampling uses the parameter `"top_k": 50` and `"top_p": 0.95`.

All model configurations are combined with the three models CodeLlama, MistralHermes and Magicoder. Each evaluation of one of the three runs "Default", "Masked" and "Use completion" runs one dataset evaluation for all model configurations, with and without the approach, totaling to $3 * 6 * 3 * 2 = 108$ dataset evaluations. The repeated dataset evaluations of the baseline are used to average results.

5.2 Research Questions

This section lists the research question for the thesis. The research questions for the approach are directly derived from the requirements of the thesis listed in Table 2. Each research question corresponds to one or several requirements. The following research questions are posed to evaluate the approach in the results chapter:

RQ1 How does the approach affect the use of dependencies?

RQ2 How does the approach handle different types of dependencies?

RQ3 How well can the approach be extended to other languages?

RQ4 How much overhead does the approach introduce?

RQ1 How does the approach affect the use of dependencies?
The first research question combines questions derived from the requirements **R1** and **R2**. It aims to understand how the models use of dependencies changes when applying the approach. This research question is further divided into three subquestions:

RQ1.1 In which scenarios does regular code completion fail to use dependencies correctly?

RQ1.2 Which scenarios does the approach improve best and under which conditions?

RQ1.3 Why does the approach not improve the results in other cases?

***RQ1.1** In which scenarios does regular code completion fail to use dependencies correctly?*
While further subquestions focus on possible improvements using the approach, this question starts with the current state of code completion. It goes beyond simply evaluating the accuracy of the baseline, but aims to identify concrete scenarios where the baseline fails to use dependencies correctly. Further research subquestions shall analyze how the approach impacts these scenarios.

In the evaluation, only the results of the baseline are considered. The results are mainly grouped by the scenario of the task. To further distinguish trends between model configurations and models, the results are also grouped by each model and different configurations. For each scenario, failing tasks are analyzed, concretely identifying the issues.

Additionally, Copilot is used to evaluate DependencyEval, obtaining further baseline results.

***RQ1.2** Which scenarios does the approach improve best and under which conditions?* After identifying issues with the baseline, the approach is applied to improve the results. This research subquestion focuses on the cases where the approach improves the results the most. It aims to understand why the improvement occurs and under which conditions the approach is most effective. Next to analyzing differences in the prompt and logits manipulation, also differences between model configurations and capabilities are considered.

Both the results of the baseline and the approach are considered. Similar to **RQ1.1**, the results are mainly grouped by the scenario of the task, and secondly by each model and different configurations. The difference is that now only the successful tasks are analyzed.

***RQ1.3** Why does the approach not improve the results in other cases?* In contrast to the previous research subquestion, this subquestions focuses on the remaining cases where the approach does *not* improve the results, or even worsens them. It aims to understand why the approach fails, to identify shortcomings and limitations of current models, model configurations, and the approach itself.

The remaining tasks in the experiments are analyzed, focusing on the differences between the baseline and the approach.

RQ2 How does the approach handle different types of dependencies?
The second research question focuses on the dependencies themselves. It is derived from the requirements **R3**. In contrast to **RQ1**, which analyzes code scenarios of DependencyEval, this research question discusses the performance of the approach on used dependencies, based on different attributes of the dependency. It analyzes and discusses the attributes dependency size and documentation availability. This research question is further divided into two subquestions:

RQ2.1 How does the availability of documentation affect the results?
RQ2.2 How does the approach handle differently sized code bases?

***RQ2.1** How does the availability of documentation affect the results?* The first subquestion analyzes the use of the information source, code documentation. Can the approach improve the results when almost no documentation is available? Does the approach struggle with too much documentation?

***RQ2.2** How does the approach handle differently sized code bases?* Similar to **RQ2.1**, this research question aims to understand the impact of the size of the code base on the performance of the approach. Are there any distinguishable patterns?

RQ3 How well does the approach generalize to other languages?
The third research question focuses on the generalizability of the approach. It is derived from the requirements **R4**. The approach should be generalizable to all programming languages. This research question investigates if the approach is extensible to other languages and if there any limitations. How large is the effort to add support for a new language? The form of the research question will be a discussion. Furthermore, it will quickly analyze the state of the current implementation.

RQ4 How much overhead does the approach introduce?
Lastly, this thesis will analyze the performance overhead introduced by the approach. After discussing the theoretical overhead, the results section will display the runtime compared against the baseline.

6 Results

After introducing the experiment setup, run experiments, and research questions, this section lists the results for each research question. If not specified otherwise, percentage differences are given in percentage points.

RQ1 How does the approach affect the use of dependencies?

RQ1.1 *In which scenarios does regular code completion fail to use dependencies correctly?* Figure 8 displays the average completion rate using the baseline (approach is disabled). The three models CodeLlama, MistralHermes and Magicoder are able to solve 34.52% of the tasks in a traditional sense (fully and partially solved). When considering the used approach of the model, now only 6% are solved (fully solved). In comparison, Copilot is the superior baseline model, solving 52% in a traditional sense and still solving 8% when considering the approach. As can be seen, there is a need for the approach to transform the 28.8% partially solved tasks (over all models) into fully solved tasks, by using the correct approach.

To note is that Magicoder and MistralHermes perform more errors than CodeLlama and Copilot, but at the same time have less unsolved tasks. Possibly Magicoder and MistralHermes fail harder on some tasks. Additionally, code models fail to complete the correct solution, even when directly prompted. Copilot fails to use the correct method, even when supplied with a code comment stating otherwise (Listing 81). This further shows why approaches only modifying the prompt do not suffice.

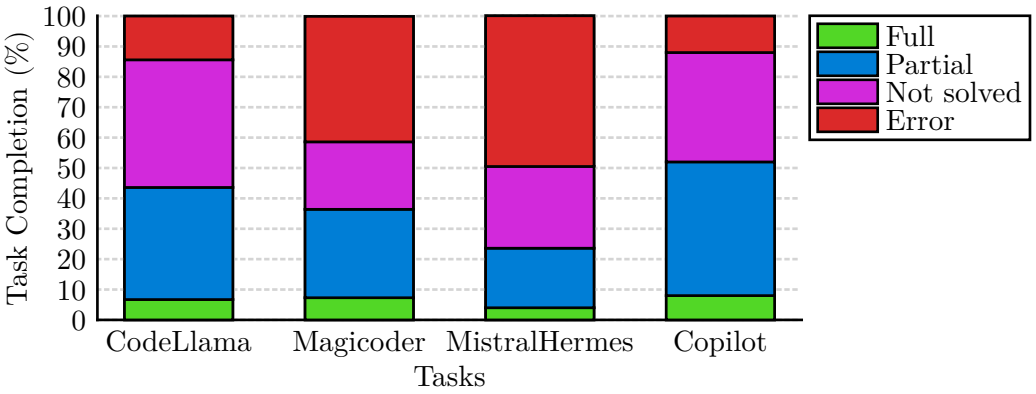


Figure 8: Overview over the baseline evaluation. The first three models are averaged due to the use of sampling. The full stats are found in Table 17.

After comparing the baseline performance of different models, this paragraph compares baseline results for different model configurations. Figure 9 shows baseline results for different model configurations. The best performing configurations use 3 beams and sampling. Only using a single beam performs particularly bad, with 0% fully solved tasks and only 10.67% partially solved tasks. This is partially in line with the observations by Austin et al. [35] (Section 2.6), which noted that sampling performs best. While they concluded that greedy decoding performs better than beam search, they did not compare sampling beam search to other model configurations. This paper notes that the high repetition penalty (1.3 for CodeLlama), combined with sampling may have led to the high beam search performance, which suffered from high repetitions in Austin et al's. work.

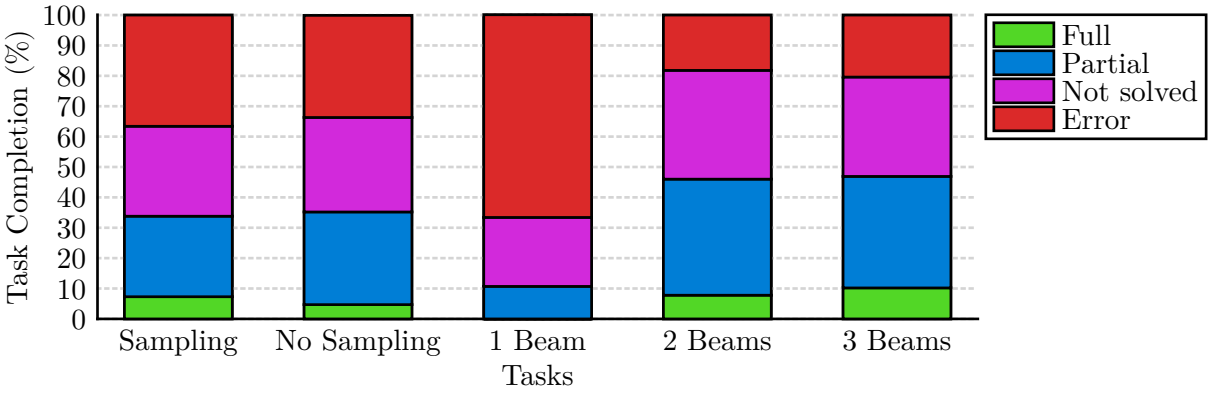


Figure 9: Overview over the baseline evaluation using different model configurations. Copilot is not considered. The full results are found in Table 18.

Next, Figure 10 compares the three approach configurations. As expected, the three configurations only display a negligible difference. As fully expected, the approach configuration does not affect the generation when the approach is disabled. Thus, the plot displays the average task completion over the baseline results, as the approach configuration should not affect the baseline.

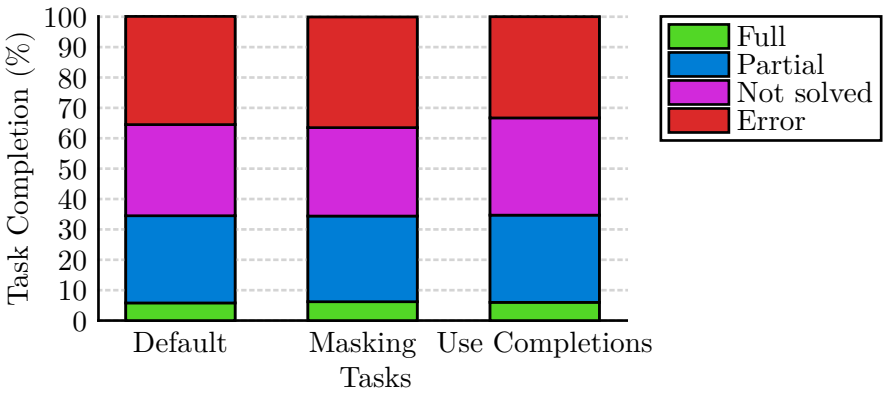


Figure 10: Overview over the baseline evaluation using different approach configurations. The full results are found in Table 19.

The next table, Table 7, shows the results for each task. The tasks are sorted from best to worst solved tasks (sorted by fully solved, partially solved, not solved, then error descending). As can be seen, only 24%(6) of all different tasks are ever fully solved by the baseline, while 76%(19) are at least partially solved once[1].

The gold solution to these first six tasks pytorch_1, bidict_2, rich_2, pandas_1, dateutil_1 and polars_1 are found in Listing 66, Listing 54, Listing 70, Listing 61, Listing 55 and Listing 62. pytorch_1 is the highest fully solved task with 61.82%. This task includes changes from 2018, and is one of the tasks with the oldest changes. It has been included to check the behavior of using dependencies with features deprecated in the "far past". As the baseline evaluations are able to solve the task consistently, it can be assumed that similar code can be found in many training data samples, as developers have migrated away from the previous deprecated parameter to the correct parameter reduction.

[1]The exact distribution of task results by Copilot are found in Table 21. The results are in line with Table 7.

The five other tasks feature recent (2024) changes. It is theoretisized that the baseline models solve these tasks due to their easily predictable result, due to e.g. the information found in the documentation string provided in each task, which is especially true for bidict_2.

The task dateutil_1 tests the behavior when facing changed import behavior. The recent modification in 2024 added lazy import of submodules. Submodules must not be explicitly imported, but can be imported implicitly through the base module. E.g.: dateutil.tz can be used by only importing dateutil. As other behavior around the task has not changed over time, the model baseline still solves the task in some cases, not explicitly importing the submodule.

The next 60%(15) tasks (up and including textual_2) are not fully solved, but at least partially solved. These tasks mainly include tasks with recent deprecations. The baseline still predicts old behavior, passing the output correctness test, but failing the approach correctness test. A prime example are the pydantic examples, as introduced in previous chapters.

The last group of tasks are never solved by the baseline. This group includes especially complex tasks that may require multiple steps, or completely unknown packages, that do not see a widespread use. The three models especially struggle with unknown dependencies, or unknown features in better known dependencies, such as theflow_1 Listing 76, tsv2py_1 Listing 76 and sqlalchemy_1 Listing 73. Tasks with recent deprecations in established dependencies are more easily solved, such as pydantic examples. The easiest tasks feature other changes, such as changes to imports and code changed in the further past.

The first group of tasks fully solved by the baseline serve to check if the approach worsens the results. The second group of tasks partially solved by the baseline serve to check if the approach can improve the results. The last group of tasks never solved by the baseline serve to check if the approach can solve tasks that are not solvable by the baseline, and to find further interesting results.

Table 7: Baseline evaluation results for each task. The table is sorted from best to worst solved tasks. Copilot results are included.

Name	Fully solved	Partially solved	Not solved	Error
pytorch_1	63.64%	12.73%	0%	23.64%
bidict_2	43.64%	20%	14.55%	21.82%
rich_2	20%	7.27%	40%	32.73%
pandas_1	14.55%	9.09%	16.36%	60%
dateutil_1	5.45%	32.73%	34.55%	27.27%
polars_1	3.64%	0%	65.45%	30.91%
pytorch_3	0%	72.73%	3.64%	23.64%
pytorch_2	0%	70.91%	0%	29.09%
dotted_1	0%	69.09%	5.45%	25.45%
pydantic_1	0%	67.27%	12.73%	20%
pydantic_3	0%	63.64%	10.91%	25.45%
sklearn_2	0%	56.36%	18.18%	25.45%
rich_1	0%	52.73%	25.45%	21.82%
pydantic_2	0%	49.09%	23.64%	27.27%
numpy_1	0%	45.45%	7.27%	47.27%
textual_1	0%	30.91%	41.82%	27.27%
dotted_2	0%	25.45%	52.73%	21.82%
fastapi_1	0%	16.36%	41.82%	41.82%
bidict_1	0%	14.55%	52.73%	32.73%
emoji_1	0%	1.82%	74.55%	23.64%
textual_2	0%	1.82%	72.73%	25.45%
sqlalchemy_1	0%	0%	80%	20%
tsv2py_1	0%	0%	67.27%	32.73%
sklearn_1	0%	0%	0%	100%
theflow_1	0%	0%	0%	100%
Average	6.04%	28.8%	30.47%	34.69%

As could be seen in previous figures, choosing the right configuration changes the results drastically. Even more so when combining the different choices. While the averaged baseline evaluations often do not complete more than 10% of tasks fully, this cannot be said for individual evaluations. Figure 11 displays the top 5 best configurations. It is directly apparent, that those individual evaluations only use sampling and 2-3 beams. This is as expected and in line with Figure 9. Additionally, the best configurations produce substantially less errors and many partially solved tasks. While a fully solved rate of up to 20% is more impressive, it is not sufficient and leaves much room for improvement to the approach.

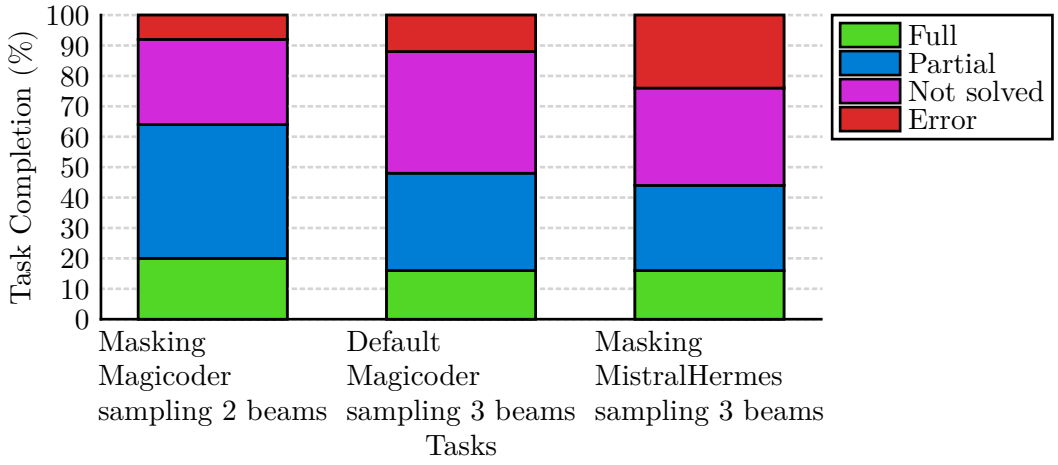


Figure 11: Overview over the top 3 baseline configurations. The full results are found in Table 20.

In summary, all configurations fail to use dependencies correctly, except for a few tasks that are very similar to training data. Especially scenarios that include recently deprecated code are affected. While other tasks like theflow_1 are not solved by the baseline, this is likely due to proportionally high difficulty of the task.

RQ1.2 *Which scenarios does the approach improve best and under which conditions?* This section now looks at the difference between the results using the approach compared against the baseline. In particular, this section looks at improvements in different scenarios. All improvement percentages are relative to the total tasks, **not** the category (like fully solved).

The first table, Table 8, shows the differences between baseline and approach for the three models. CodeLlama is most affected by the approach, improving all fully solved tasks by 7.56%. This is in stark contrast to MistralHermes, which only shows improvements in fully solved tasks of 1.56%.

Table 8: Effects of using different models for evaluation of the approach, compared to the baseline.

Name	Fully solved	Partially solved	Not solved	Error	Unchanged
Codellama	+7.56%	−16.22%	−4.22%	+12.89%	79.56%
Magicoder	+4.22%	−17.56%	−3.56%	+16.89%	78.89%
MistralHermes	+1.33%	−6.67%	−7.33%	+12.67%	86%

In similar contrast are the results when comparing different model configurations. Table 9 displays the differences between the sampling and non sampling model configurations. The sampling model configuration using the approach shows the most improvements of 7.85%, in contrast to the 1.04% when not using sampling. Similar differences are seen in Table 10 between the best configuration using 2 beams (+8%), versus the 1 beam configuration only showing improvements of 1.78%.

Table 9: Effects of using sampling vs. not using sampling on the evaluation results, compared against the baseline.

Name	Fully solved	Partially solved	Not solved	Error	Unchanged
No Sampling	+1.04%	−15.41%	−5.93%	+20.3%	78.67%
Sampling	+7.7%	−11.56%	−4.15%	+8%	84.3%

Table 10: Effects of using different beam sizes on the evaluation results, compared against the baseline.

Name	Fully solved	Partially solved	Not solved	Error	Unchanged
1 Beam	+1.78%	−1.78%	−3.33%	+3.33%	94.89%
2 Beams	+8%	−18.22%	−7.11%	+17.33%	74.67%
3 Beams	+3.33%	−20.44%	−4.67%	+21.78%	74.89%

The next table Table 11 moves on to comparing the three approach configurations “Default”, “Masking” and “Use Completions”. The configuration with the most impact is the “Default” configuration.

Table 11: Effects of using different approach configurations on the evaluation results, compared against the baseline.

Name	Fully solved	Partially solved	Not solved	Error	Unchanged
Default	+5.56%	−11.78%	−2.44%	+8.67%	85.78%
Masking	+4.89%	−10.44%	−2.89%	+8.44%	86.67%
Use Completions	+2.67%	−18.22%	−9.78%	+25.33%	72%

After looking at different conditions optimal to improve the tasks, Table 12 looks at differences in concrete tasks / scenarios. The tasks are sorted from best to worst solved tasks.

Table 12: Difference between the baseline and the approach evaluation results for each task. The table is sorted from best to worst solved tasks.

Name	Fully solved	Partially solved	Not solved	Error	Unchanged
pydantic_1	+44.44%	−51.85%	+1.85%	+5.56%	48.15%
textual_1	+25.93%	−31.48%	−24.07%	+29.63%	44.44%
pydantic_3	+25.93%	−44.44%	+12.96%	+5.56%	55.56%
rich_2	+24.07%	−3.7%	−37.04%	+16.67%	59.26%
dateutil_1	+14.81%	−1.85%	−27.78%	+14.81%	70.37%
pydantic_2	+9.26%	−31.48%	+7.41%	+14.81%	68.52%
pandas_1	+3.7%	+5.56%	±0%	−9.26%	90.74%
rich_1	+1.85%	−14.81%	−3.7%	+16.67%	81.48%
textual_2	±0%	+1.85%	−20.37%	+18.52%	79.63%
emoji_1	±0%	+1.85%	−24.07%	+22.22%	75.93%
sklearn_1	±0%	±0%	±0%	±0%	100%
theflow_1	±0%	±0%	±0%	±0%	100%
bidict_1	±0%	±0%	−7.41%	+7.41%	92.59%
sqlalchemy_1	±0%	±0%	−9.26%	+9.26%	90.74%
tsv2py_1	±0%	±0%	−18.52%	+18.52%	81.48%
bidict_2	±0%	−7.41%	+1.85%	+5.56%	92.59%
dotted_2	±0%	−9.26%	−14.81%	+24.07%	75.93%
fastapi_1	±0%	−16.67%	−9.26%	+25.93%	74.07%
dotted_1	±0%	−20.37%	+1.85%	+18.52%	79.63%
numpy_1	±0%	−25.93%	+18.52%	+7.41%	74.07%
pytorch_2	±0%	−27.78%	+1.85%	+25.93%	72.22%
pytorch_3	±0%	−31.48%	+9.26%	+22.22%	68.52%
sklearn_2	±0%	−55.56%	+46.3%	+9.26%	44.44%
polars_1	−3.7%	±0%	−31.48%	+35.19%	64.81%
pytorch_1	−37.04%	+27.78%	±0%	+9.26%	62.96%
Average	+4.37%	−13.48%	−5.04%	+14.15%	73.93%

After looking at the tasks, three groups emerge:
1. A few tasks heavily improved by the approach,
2. Many tasks unaffected by the approach,
3. A few tasks worsened by the approach.

The first group shows which scenarios the approach improves best. These unique tasks mainly include the pydantic tasks and the textual_1 tasks, which correspond to the second baseline task group. In the baseline those tasks mainly had a high partial solved rate, as the dependencies were recently changed and included simple deprecations of methods and function parameters. The approach clearly improves such examples the best, changing the single deprecated method or parameter to an alternative.

Table 13: Overview over the top 3 configurations improving the baseline using the approach.

Name	Fully solved	Partially solved	Not solved	Error	Unchanged
Default CodeLlama sampling 2 beams	+20%	−16%	−12%	+8%	72%
Masking MistralHermes sampling 2 beams	+16%	−8%	±0%	−8%	84%
Masking CodeLlama sampling 2 beams	+16%	−12%	−4%	±0%	84%

To summarize, only certain configuration combinations improve the results. Especially CodeLlama using sampling and two beams, using the "Default" approach configuration, shows the most improvements. Additionally, only a handful of tasks can be improved. Those are the core tasks in which a method or function parameter was recently deprecated. Other tasks do not show many or any improvements. To note is that not only does sampling beam search perform best of the baseline results, but is also best suited for the approach.

***RQ1.3** Why does the approach not improve the results in other cases?*

This section first looks at failing examples in previous figures and analyzes why the approach worsens the results in these cases. It e.g. continues the notes on the different task groups from the last section.

As introduced before, the second task group mainly includes tasks unaffected by the approach. The main observation is that many of these tasks are some of the worst performing baseline tasks. E.g. sklearn_1 and theflow_1 have a 0% completion rate in the baseline and show no improvements using the approach. These tasks seem to be unaffected due to their proportionally high difficulty, as providing additional context does not improve the base model capabilities. The only outlier being bidict_2, which is one of the best performing tasks in the baseline, but is not improved by the approach. This task is likely to be too simple for the approach to improve, as the baseline already solves it consistently.

The last group of tasks worsened by the approach mainly correspond to some of the best performing baseline tasks. This is especially true for the pytorch tasks, with pytorch_1 the main example. The conclusion is that the simpleness of the tasks is worsened by the approach, as the approach introduces additionaly complexity to the completion, which in some cases causes the generation to end in faulty code, where the model without the approach would not have been guided towards.

When looking at different configurations again, it becomes clear that limiting the search space by using only a single beam, or not using sampling, worsens the results. Exploring many different combinations of completions affected by the approach reduces the amount of errors.

There are several cases where using the approach led to a model confusion. The following four examples show different types of confusions occuring:

In the first example Listing 46, the approach guides the generation towards the correct method model_dump. The error lies in the uneeded parameter by_alias, with the wrong value "...". As can be seen in the signature of model_dump (Listing 78), the parameter by_alias accepts a boolean. The completed code uses the wrong value, the ellipsis constant.

```python
from typing import Any, Dict
from pydantic import BaseModel

class User(BaseModel):
    name: str
    email: str
    age: int

def convert_user_to_dict(user: User) -> Dict[str, Any]:
    """Convert the user into a Python dictionary.

    Args:
        user (User): Pydantic user model

    Returns:
        Dict[str, Any]: User attributes as a Python key value mapping
    """
    return user.model_dump(by_alias=...)
```

Listing 46: Example of the approach confusing the model to use a wrong value for an unneeded parameter. The approach guided the generation towards the correct method model_dump.

In the next example Listing 47, now the correct function and parameter were chosen. For comparison, the function signature can be found in Listing 79. The confusion lies in Literal. Literal is a type class representing a literal value, to be specified directly. This can be seen in the default value set in the function signature. In contrast, the completed code tries to construct the parameter value using the Literal class, erronously predicting that it is a real class and not a type hint.

```python
# default/codellama_7b_greedy/textual_1.py
from textual.widgets import TextArea

def create_textual_text_area_with_indent() -> TextArea:
    """Create a TextArea widget, which indents its content when tab is pressed.

    Returns:
        TextArea: New instance of TextArea with indentation on tab press
    """
    return TextArea(tab_behavior=Literal["indent"])
```

Listing 47: Example of the approach confusing the model to use a wrong type for a parameter value. The approach guided the generation towards the correct method text_area, but was confused by the Literal type hint.

The third example Listing 48 goes one step further. The used function and parameters use the correct approach, similar to the correct result Listing 59. The difference is the

generation of a type hint for the return type, which is incorrect outside of function definitions.

```python
# default/codellama_7b_greedy_beams_3/fastapi_1.py
from contextlib import asynccontextmanager
from fastapi import FastAPI

def startup(app: FastAPI):
    app.state.startup = True

def shutdown(app: FastAPI):
    app.state.shutdown = True

def create_fastapi_app() -> FastAPI:
    """Create a new FastAPI app which calls the lifespan functions startup and shutdown.

    Returns:
        FastAPI: New FastAPI instance
    """
    return FastAPI(on_startup=startup, on_shutdown=shutdown) -> FastAPI
```

Listing 48: Example of the approach confusing the model to use a wrong type hint for a return value. The approach guided the generation towards the correct method app, but was confused by the type hint.

Next, Listing 49 calls the correct method, but is confused by the method parameters. While the clear_meta_and_links takes no parameters (except self), the completed code passes the value True. This can be explained by the method being decorated with @lru_cache, which proxies the method. This destroys the original function signature [66], leading to the function signature found in Listing 80.

```python
# default/codellama_7b_greedy_beams_2/rich_1.py
from rich.style import Style

def clear_style(style: Style) -> Style:
    """Obtain a copy of style with all meta and links removed.

    Args:
        style (Style): target style

    Returns:
        Style: target style without meta and links
    """
    return style.clear_meta_and_links(True)
```

Listing 49: Example of the approach confusing the model to use invalid parameters for a method. The approach guided the generation towards the correct method clear_meta_and_links, but was confused by the method signature.

And lastly, Listing 50 completed the correct code, followed by an invalid postfix. The model is generally confused and instead of stopping the generation, continues in an invalid format.

```
# default/mistralhermes_codepro_7b_sampling/rich_2.py
from rich.prompt import Prompt

def create_case_insensitive_prompt(text: str) -> Prompt:
    """Create a prompt instance, providing the text and using case insensitivity.

    Args:
        text (str): prompt text

    Returns:
        Prompt: created prompt
    """
    return Prompt(text, case_sensitive=False)</s>
    123456789
``
```

Python

Listing 50: Example of the correct code followed by an incorrect postfix, due to a bad format.

In summary, most errors are caused due to issues with type information. Some other errors are caused due to wrong formatting.

After looking at general causes for erroneously generated code, this section will analyze the two distinct error classes of runtime and evaluation errors. It distinguishes between two types of errors:

1. Generation errors, which occur during the generation of the code due to issues in the runtime.
2. Evaluation errors, which occur during the evaluation of the generated code.

The first table, Table 14, displays all occuring model generation errors when using the approach. Only a single error occurs, the error OutOfMemoryError. This error is caused by the generation trying to consume more than 24GB GPU memory (full GPU memory of 1 GPU used for generation). When looking at the used approach configuration, it becomes that the error only occurs when the “use completion” configuration is used. The “Use Completion” configuration enables the approach to directly infer a method name using a secondary model generation. It seems that the secondary generation consumes additional memory over the total capacity of the used GPU. This seems to be an issue especially when using beam search, which consumes more resources. With greedy decoding the error does not occur. Additionally it only occurs for certain tasks. This leads to the conclusion that the high GPU memory use is due to an unbounded generation, likely due to model repetition. Using a model configuration or no beam search or other tasks yields higher quality results, possibly terminating the generation quicker, consuming less resources. To note is that no runtime errors occured when using MistralHermes.

Table 14: All runtime errors occuring during the model generation when using the approach. The card displays the in total 17 errors, grouped into different categories, such as the models, model configurations, etc..

Group Name	Group Occurence
Total	17
Models	
CodeLlama	52.94%
Magicoder	47.06%
Model Configurations	
3 beams	52.94%
2 beams	47.06%
Sampling	52.94%
No sampling	47.06%
Approach Configurations	
Use Completions	100%
Tasks	
pytorch_3	47.06%
pytorch_2	47.06%
textual_2	5.88%
Errors	
OutOfMemoryError	100%

The second table, Table 15, displays all evaluation errors occuring when using the approach, which did *not* occur in the baseline. The values here are different to errors in previous sections, as runtime errors are not considered. Additionally, the effect of the approach solving a previously erroneous task is not considered, which is why numbers vary, compared to previous sections. The first note is that errors are split evenly between different model configurations. This indicates that while beam search is suited for the baseline *and* the approach, that if the model fails, it fails hard, without (almost) any difference between model configuration. Applying beam search with sampling may improve the approach without having an effect on failing examples. Additionally, beam search using 2 beams slightly outperforms other configurations.

When looking at approach configurations, using the "Masking" approach configuration leads produces the most new evaluation errors, indicating that slightly nudging the model logits during generation prevents cases where due to the hard masking helpful alternatives are removed, which would have otherwise been chosen due to sampling.

Now the different tasks in which new errors occur due to the approach can be compared to tasks, for which the average amount of errors increased. It becomes clear, that entries with a high amount of errors in the baseline are either not listed, or have a low score. As those entries already have many errors, almost no new errors can occur. This is the case for e.g. "theflow_1".

The next note is that “pandas_1” has a high increase in (absolute) errors, even though the average amount of errors sinks, as seen in Table 12. This is likely due to the approach either strongly guiding the generation to a solution, thus reducing the error average, or strongly mistakenly guiding the generation towards a new error.

When looking at error categories, unexpectedly, a large portion of errors are due to incorrect indentation. After reviewing examples, these are often due to Magicoder incorrectly indenting the line with 2 instead of 4 whitespace. This error was mainly found when using greedy decoding and rarely with beam search.

Table 15: All evaluation errors occuring when using the approach, which did *not* occur in the baseline. The card displays the in total 80 errors, grouped into different categories, such as the models, model configurations, etc..

Group Name	Group Occurence
Total	80
Models	
MistralHermes	58.75%
CodeLlama	23.75%
Magicoder	17.5%
Model Configurations	
3 beams	36.25%
1 beams	35%
2 beams	28.75%
Sampling	50%
No sampling	50%
Approach Configurations	
Masking	37.5%
Default	32.5%
Use Completions	30%
Tasks	
pytorch_1	21.25%
pandas_1	13.75%
bidict_2	8.75%
numpy_1	8.75%
polars_1	7.5%
bidict_1	3.75%
textual_2	3.75%
sklearn_2	3.75%
fastapi_1	3.75%
tsv2py_1	3.75%
pydantic_3	2.5%
pydantic_2	2.5%
pydantic_1	2.5%
dotted_2	2.5%
rich_2	2.5%
pytorch_2	2.5%
rich_1	1.25%
dotted_1	1.25%
pytorch_3	1.25%
dateutil_1	1.25%
sqlalchemy_1	1.25%
Errors	
SyntaxError	26.25%
IndentationError	8.75%
NameError	2.5%

In summary, some tasks like "theflow_1" in DependencyEval are very hard, leading to an 100% error rate in the baseline and approach. Additionally, the added complexity of the approach causes some simple tasks to fail, such as "pytorch_1". Next to such evaluation errors, the model Magicoder causes indentation errors when using greedy decoding. Additionally, the approach configuration "Use Completions" causes a high GPU usage, sometimes causing out of memory exceptions at runtime.

RQ2 How does the approach handle different types of dependencies?

To answer this question, in particular the two subquestions **RQ2.1** and **RQ2.2**, this section presents Table 16. The table shows the size of the dependencies used in DependencyEval, as counted using pygount. The number of pure lines of code, comments and empty lines are counted. Additionally, the documentation percentage relative to the lines of code is calculated.

Table 16: Size of dependencies, as counted using pygount. Documentation percentage is relative to the lines of code, thus excluding empty lines, etc.

Dependency	Lines of Code	Documentation Percentage
bidict	111	41.08%
scipy	222189	35.37%
tsv	146	31.56%
sqlalchemy	107258	31.03%
dateutil	4235	29.26%
numpy	163415	26.17%
textual	27344	26.03%
polars	9064	24.09%
emoji	287	24.03%
theflow	2734	22.71%
pydantic	21116	21.18%
torch	726475	18.85%
rich	10965	18.3%
pandas	327097	17.96%
dotted	473	11.42%
fastapi	4807	3.67%

***RQ2.1** How does the availability of documentation affect the results?*

The results are inconclusive using the current evaluation method. While Table 12 shows changes when applying the approach to different dependencies, it is hard to draw conclusions on the effect of documentation. Methods such as counting lines of documentation and comparing the results are simple, but not a good indicator. To correctly analyze the effect of documentation, the amount of available documentation in the retrieved context must be analyzed. This leads to another metric, the amount of documentation available per code item, such as a method. This greatly increases the score for different libraries in comparison to the simple lines of code method, as the length of functions becomes irrelevant. The upside of such a new method would be that it would correctly find problems imposed due to too little or too much documentation, such as context length issues.

After reviewing documentation of different libraries by hand, it becomes clear that especially libraries like Numpy have extensive documentation, not represented by the documentation percentage in Table 16. Each function documentation includes a short description, parameter documentation and even often notes and examples. This extensive documentation has proven to be tricky to inject into the model prompt, which is why the approach has chosen to reduce the function documentation to the short descriptive sentence, using the documentation parser `docstring_parser`. While this has solved the issue of too much documentation, analyzing the effects of including notes and examples would be desired.

***RQ2.2** How does the approach handle differently sized code bases?* In a similar manner, the results are inconclusive using the current evaluation method. While lines of code are a good indicator for the total size of the dependency, it is not a good indicator for the size of the dependency in the context of the task. As the model is guided using a language server, the model navigates on a subset of the dependency, being presented a list of completions for the current context. When imagining dependencies as a code item tree, where the dependency is the root, submodules are branches, files form branches in the next level, followed by classes, methods and finally parameters, the branching factor becomes important for the approach. The branching factor determines, how many entries a code completion request to the language server returns. This is especially important for the approach, as the model is guided by the completions and too many completions or too few may pose an issue. This proves difficult to measure over the whole repository, as the language server would need to navigate the whole repository to find the branching factor.

In practice, how the approach handles different amounts of completion entries depends on the configuration used. When using the "Use Completions" configuration, completion entry information is used to determine the next code symbol, such as a method call. Instead of only guiding the next token, the model is asked separately to directly predict the whole method call, based on all available method calls, while being noted about deprecations. While this configuration handles small to medium amounts of completion items, it fails for especially large amounts.

In any case, all completion items are used to rerank the logits of the predicted next token in the logits processor. This approach has proven to handle any amounts of completion items in practice. In summary, the measured results are inconclusive, empirically obtained results are positive, but further complex testing is necessary.

RQ3 How well does the approach generalize to other languages?
While the approach theoretically generalizes well to other languages, as it used the language independent LSP, in practice there are many restrictions. The first restriction is that a language server must be available for use. This is not the case for every programming language[1]. If no language server is available, creating a new language server takes a significant amount of effort, which is often undertaken by the community. Even if a language server is available, it may not support the needed functionality of the approach, such as retrieving completion items and a signature help.

Additionally, generalizing over project setups may prove difficult. Language servers are often customizable by passing options from the client to the server, controlling behavior such as finding project dependencies. Adding support for a new language includes understanding and setting these options, such that the language servers of a potential new language correctly determine all project dependencies. At last, the current implementation has assumptions about common code structure. This will prove wrong for some other languages, especially esoteric languages like Brainfuck. In practice these assumptions may be removed and the implementation improved, which will require further effort.

[1]E.g. no publicly available language server was found for the ABAP programming language.

In summary, the approach current generalizes well over similar and widely used languages, which have a language server. Adding support for other esoteric programming languages is possible, but requires a significant effort.

RQ4 How much overhead does the approach introduce?

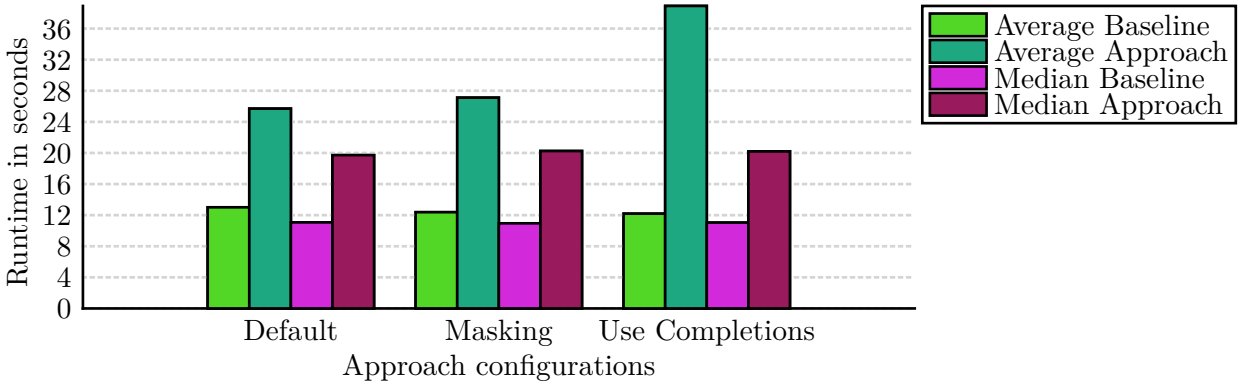


Figure 12: Average runtime in seconds per task, comparing the baseline and the approach. When comparing the median, the approach only less than doubles the runtime. When looking at the average runtime, it becomes clear that the approach incurs a larger penalty, sometimes tripling the runtime.

To evaluate the runtime overhead, Figure 12 compares the runtime of the approach to the baseline. As can be seen, the approach less than doubles the runtime in most cases. When looking at the average runtime, it becomes clear that a few code completions take significantly longer, on average it takes thrice the baseline runtime.

The used approach configuration will not impact the approach runtime under a best-case scenario. Each configuration causes the median runtime to rise from about 10 seconds to 20 seconds, doubling the runtime. Under a worst-case scenario, performance is most impacted by the "Use Completions" configuration. This is most likely due to the second generation performed when e.g. directly predicting the next possible method. When the model responds with an additionally long message, it adds to the total runtime quickly.

The approach only incurs a small performance penalty, as the static analysis performs many performance heavy tasks, which are heavily optimized, from which the approach may profit. Due to the IBMS further runtime overhead may be saved, compared to typical multi-step generation meta-strategies. The current implementation is not fully optimized towards a short runtime. Through heavier caching, a lower runtime may be achieved. Especially in beam search configurations, separate beams generate similar code, retrieving similar code context, which may be shared between beams. The current implementation does not share such data between beams.

7 Conclusion

7.1 Recap

LLMs are widely used in the field of code completion. Tools like Copilot find a widespread use, reducing the need to write boilerplate code. While these tools are useful, they especially struggle with rapidly changing dependencies. Adapting the model to changed dependencies requires significant resources. Alternatives use the inherent model capabilities to inject accurate dependency information. Previous approaches often retrieve code context using RAG and inject it into the prompt.

The goal of this thesis was to explore different ways to guide code completion, injecting accurate code context at runtime, additionally reducing model runtime compared to multi-step generation approaches. Accurate code context is retrieved using IDE tooling, retrieving code documentation, completion items and signature information. This enables the approach to inject context for all code, from the repository itself to external dependencies. IDE tooling is used through the LSP, retrieving context for any supported programming language. This allows the approach to be language-agnostic.

Then MGD is explored, observing the model at runtime to solve the "context retrieval problem". Observing the model at runtime allows the approach to provide relevant context online, reducing the need for multi-step generation approaches and reducing runtime. To inject the retrieved context, two different mechanisms are explored. The first mechanism adds context to the prompt. It is suited for unstructured context, such as code documentation of completion items and guides the model softly. As shown, the model may be biased towards generating deprecated code, even when the prompt includes code context stating otherwise (Listing 81). In these cases the model must be guided through other means - the second mechanism. The second mechanism modifies the model logits at runtime, reranking the token id scores of the next predicted token. This mechanism is best suited for completion items and signature help information, as that context may be used to directly influence the next token. Next, tokens of suggested deprecated completion items are used to downrank the model logits, while next tokens of other suggested completion items are used to uprank the model logits.

The approach then combines both mechanisms, softly guiding the model through the prompt and reranking the logits at runtime. This enables a more robust approach, as the hard logits reranking solves the issue of the model generating deprecated code, while the soft context injection guides the model through the prompt. The thesis then introduces the Interrupt-Based Meta-Strategy (IBMS), a method which injects context into the prompt during generation, removing the need for multi-step generation approaches. It is designed to work with any context, working with retrieved code context, but compatible with other tasks. IBMS is capable of removing injected context, enabling injection of scoped context.

To evaluate the approach, the thesis introduces the dataset DependencyEval, a dataset containing handcrafted Python code completion tasks with external dependencies. Each task includes the exact dependencies, including the Python version. Additionally, it features an evaluation pipeline to automatically evaluate the approach in different configurations, collecting different metrics and plotting results. Each task includes two unit tests, for

testing the functional correctness and the approach correctness of the generated code. The approach correctness verifies that no deprecated code was used. The dataset can be built from individual code files, simplifying the process of creating new tasks.

Lastly, the thesis evaluates the approach on DependencyEval. The results show that the approach successfully guides the model away from deprecated code, using the correct alternatives. An observation is that choosing the correct approach and model configurations and code models have drastic effects on the results. Models using beam search with sampling perform the best. Even so, the high complexity of the approach increases the difficulty to make it robust. While the provided implementation improves the robustness of the approach using several mechanisms, the model still struggles with several scenarios, such as type hints that include literals.

In conclusion, this thesis explores and combines many different and new areas of code completion and model controlling. It shows how hybrid approaches can complement each other, providing a more robust and accurate code completion approach. It leaves many areas of optimizations open, which will be discussed in the final section.

7.2 Outlook

The thesis combines many topics from static analysis, to prompt injection, to logits manipulation, achieving a higher level of control over the model for code generation. An especial central component in the success of the model edits is the high quality information provided through the LSP. While this thesis utilizes information from completion items and the signature help, there are many more sources of information that could be used to further improve the model edits. These include the diagnostic information, which provides feedback on the already written code, such that may alter previously generated code.

Another venue for research is the focus on low resource languages. Models face difficulties in learning low resource languages, as they have fewer data to learn from. For this reason, it is of especial interest to improve the language support for these languages. Future work could focus on the improvement of code generation for low resource languages, analyzing how embedding static analysis information can improve the model's performance. It is still left to be seen, how models react to the control techniques used in this thesis, when applied to low resource languages.

Other works already focus on different means of control, such as model weight editing, which allows for the manipulation of the model's weights. Future work could combine information from static analysis through the LSP with model weight editing, similar to Imgrund (2024) [56], to further improve the control over the model.

Furthermore, future work could focus on the area of context injection. This work injects context into the system prompt, as the instruction tuned models have issues following instructions in other formats. Custom special tokens could be added to the vocabulary, which signify the start of certain embedded code context, such as the signature help. By training the model with the added tokens, and a fitting dataset, the instruction following capabilities for embedded contexts could potentially be improved for code completion.

At last, the thesis uses the novel dataset DependencyEval, to evaluate the approach used by completed code. It is the first of its kind, providing different types of tests to

distinguish between functionally correct code and code using the correct approach. Due to the high effort associated with creating the dataset, and it not being the focus of the thesis, the dataset is still small and only contains 25 samples. Future work could focus on expanding the dataset, to provide a more comprehensive evaluation of the model's performance using more samples.

Acronyms

ABC: Abstract Base Class

API: Application Programming Interface

BOS: Begin of Sequence

BPE: Byte Pair Encoding

CLI: Command Line Interface

CLM: Causal Language Model

Code LLM: Large Language Model for Code

EOS: End of Sequence

FiD: Fusion-in-Decoder

GQA: Grouped-Query Attention

IBMS: Interrupt-Based Meta-Strategy

IDE: Integrated Development Environment

IPC: Inter Process Communication

JSON: JavaScript Object Notation

JSON-RPC: JavaScript Object Notation-Remote Procedure Call

LLM: Large Language Model

LM: Language Model

LRU: Least Recently Used

LSP: Language Server Protocol

MBPP: Mostly Basic Python Problems

MGD: Monitor-Guided Decoding

MIME: Multipurpose Internet Mail Extensions

PAD: Padding

PEP: Python Enhancement Proposal

PyPI: Python Package Index

RAG: Retrieval Augmented Generation

RPC: Remote Procedure Call

RoPE: Rotary Position Embedding

URI: Uniform Resource Identifier

Bibliography

[1] S. Amann, S. Proksch, S. Nadi, and M. Mezini, "A Study of Visual Studio Usage in Practice," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Suita: IEEE, Mar. 2016, pp. 124–134. doi: 10.1109/SANER.2016.39[1].

[2] A. Ziegler *et al.*, "Productivity Assessment of Neural Code Completion," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, San Diego CA USA: ACM, Jun. 2022, pp. 21–29. doi: 10.1145/3520312.3534864[2].

[3] L. A. Agrawal, A. Kanade, N. Goyal, S. K. Lahiri, and S. K. Rajamani, "Monitor-Guided Decoding of Code LMs with Static Analysis of Repository Context."

[4] J. Goodman, "A Bit of Progress in Language Modeling." Accessed: Aug. 07, 2024. [Online]. Available: http://arxiv.org/abs/cs/0108005

[5] "Speech and Language Processing." Accessed: Jun. 04, 2024. [Online]. Available: https://web.stanford.edu/~jurafsky/slp3/

[6] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A Neural Probabilistic Language Model."

[7] "We Added 690 New Words to the Dictionary for September 2023." Accessed: Sep. 23, 2024. [Online]. Available: https://www.merriam-webster.com/wordplay/new-words-in-the-dictionary

[8] P. Gage, "A New Algorithm for Data Compression."

[9] T. Kudo and J. Richardson, "SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, E. Blanco and W. Lu, Eds., Brussels, Belgium: Association for Computational Linguistics, Nov. 2018, pp. 66–71. doi: 10.18653/v1/D18-2012[3].

[10] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving Language Understanding by Generative Pre-Training."

[11] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language Models Are Unsupervised Multitask Learners."

[12] A. Fan, M. Lewis, and Y. Dauphin, "Hierarchical Neural Story Generation." Accessed: Sep. 09, 2024. [Online]. Available: https://arxiv.org/abs/1805.04833v1

[13] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The Curious Case of Neural Text Degeneration." Accessed: Sep. 09, 2024. [Online]. Available: http://arxiv.org/abs/1904.09751

[14] T. B. Brown *et al.*, "Language Models Are Few-Shot Learners." Accessed: Jun. 12, 2024. [Online]. Available: http://arxiv.org/abs/2005.14165

[15] J. Wei *et al.*, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models." Accessed: Oct. 10, 2023. [Online]. Available: http://arxiv.org/abs/2201.11903

[16] S. Yao *et al.*, "Tree of Thoughts: Deliberate Problem Solving with Large Language Models." Accessed: Oct. 10, 2023. [Online]. Available: http://arxiv.org/abs/2305.10601

[17] " Transformers." Accessed: Jun. 18, 2024. [Online]. Available: https://huggingface.co/docs/transformers/v4.41.3/en/index

[18] "Auto Classes." Accessed: Jul. 01, 2024. [Online]. Available: https://huggingface.co/docs/transformers/v4.41.3/en/model_doc/auto

[19] "Models." Accessed: Jul. 01, 2024. [Online]. Available: https://huggingface.co/docs/transformers/v4.41.3/en/main_classes/model

[20] "Generation." Accessed: Jul. 01, 2024. [Online]. Available: https://huggingface.co/docs/transformers/v4.41.3/en/main_classes/text_generation

[21] "Utilities for Generation." Accessed: Jul. 01, 2024. [Online]. Available: https://huggingface.co/docs/transformers/v4.41.3/en/internal/generation_utils

[22] "GNU Emacs - GNU Project." Accessed: Jul. 01, 2024. [Online]. Available: https://www.gnu.org/software/emacs/

[23] "LSP Mode - Language Server Protocol Support for Emacs - LSP Mode - LSP Support for Emacs." Accessed: Jul. 01, 2024. [Online]. Available: https://emacs-lsp.github.io/lsp-mode/

[24] "Visual Studio Code - Code Editing. Redefined." Accessed: Jul. 01, 2024. [Online]. Available: https://code.visualstudio.com/

[25] "Language Server Extension Guide." Accessed: Jul. 01, 2024. [Online]. Available: https://code.visualstudio.com/api/language-extensions/language-server-extension-guide

[26] "Home - Neovim." Accessed: Jul. 01, 2024. [Online]. Available: https://neovim.io/

[27] "Lsp - Neovim Docs." Accessed: Jul. 01, 2024. [Online]. Available: https://neovim.io/doc/user/lsp.html

[28] "Zed - Code at the Speed of Thought." Accessed: Jul. 01, 2024. [Online]. Available: https://zed.dev/

[29] "Adding New Languages - Zed." Accessed: Jul. 01, 2024. [Online]. Available: https://zed.dev/docs/adding-new-languages

[30] "JetBrains IDEs: Enjoy an Exceptional Developer Experience." Accessed: Jul. 01, 2024. [Online]. Available: https://www.jetbrains.com/ides/

[31] "Language Server Protocol (LSP) | IntelliJ Platform Plugin SDK." Accessed: Jul. 01, 2024. [Online]. Available: https://plugins.jetbrains.com/docs/intellij/language-server-protocol.html

[32] "Helix-Editor/Helix." Accessed: Jul. 01, 2024. [Online]. Available: https://github.com/helix-editor/helix

[33] "Language Support." Accessed: Jul. 01, 2024. [Online]. Available: https://docs.helix-editor.com/lang-support.html

[34] "Specification." Accessed: Jul. 01, 2024. [Online]. Available: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/

[35] J. Austin *et al.*, "Program Synthesis with Large Language Models." Accessed: Jun. 02, 2024. [Online]. Available: http://arxiv.org/abs/2108.07732

[36] D. Shrivastava, D. Kocetkov, H. de Vries, D. Bahdanau, and T. Scholak, "RepoFusion: Training Code Models to Understand Your Repository." Accessed: Dec. 05, 2023. [Online]. Available: http://arxiv.org/abs/2306.10998

[37] S. Zhou, U. Alon, F. F. Xu, Z. Wang, Z. Jiang, and G. Neubig, "DocPrompting: Generating Code by Retrieving the Docs." Accessed: Jun. 23, 2024. [Online]. Available: http://arxiv.org/abs/2207.05987

[38] "DevDocs." Accessed: Jul. 04, 2024. [Online]. Available: https://devdocs.io/

[39] F. Zhang *et al.*, "RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation." Accessed: Dec. 05, 2023. [Online]. Available: http://arxiv.org/abs/2303.12570

[40] A. Eghbali and M. Pradel, "De-Hallucinator: Mitigating LLM Hallucinations in Code Generation Tasks via Iterative Grounding." Accessed: Sep. 10, 2024. [Online]. Available: http://arxiv.org/abs/2401.01701

[41] Y. Li, Y. Peng, Y. Huo, and M. R. Lyu, "Enhancing LLM-Based Coding Tools through Native Integration of IDE-Derived Static Context." Accessed: Sep. 13, 2024. [Online]. Available: http://arxiv.org/abs/2402.03630

[42] "Stack Overflow Developer Survey 2023." Accessed: Jul. 03, 2024. [Online]. Available: https://survey.stackoverflow.co/2023/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2023

[43] M. Chen *et al.*, "Evaluating Large Language Models Trained on Code." Accessed: Oct. 18, 2023. [Online]. Available: http://arxiv.org/abs/2107.03374

[44] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is Your Code Generated by ChatGPT Really Correct?."

[45] X. Du *et al.*, "ClassEval: A Manually-Crafted Benchmark for Evaluating LLMs on Class-level Code Generation." Accessed: Jul. 07, 2024. [Online]. Available: http://arxiv.org/abs/2308.01861

[46] "Styleguide." Accessed: Jul. 08, 2024. [Online]. Available: https://google.github.io/styleguide/pyguide.html

[47] "PEP 287 – reStructuredText Docstring Format | Peps.Python.Org." Accessed: Jul. 08, 2024. [Online]. Available: https://peps.python.org/pep-0287/

[48] "Unittest — Unit Testing Framework." Accessed: Jul. 08, 2024. [Online]. Available: https://docs.python.org/3/library/unittest.html

[49] "PyPI · The Python Package Index." Accessed: Jul. 08, 2024. [Online]. Available: https://pypi.org/

[50] "Serialization - Pydantic." Accessed: Jul. 08, 2024. [Online]. Available: https://docs.pydantic.dev/latest/concepts/serialization/

[51] "Textual - Home." Accessed: Jul. 08, 2024. [Online]. Available: https://textual.textualize.io/

[52] "Docker Run | Docker Docs." Accessed: Jul. 08, 2024. [Online]. Available: https://docs.docker.com/reference/cli/docker/container/run/

[53] "Tokenizer_config.Json · Codellama/CodeLlama-7b-hf at Main." Accessed: Sep. 22, 2024. [Online]. Available: https://huggingface.co/codellama/CodeLlama-7b-hf/blob/main/tokenizer_config.json

[54] "Contextlib — Utilities for with-Statement Contexts." Accessed: Aug. 05, 2024. [Online]. Available: https://docs.python.org/3/library/contextlib.html

[55] "PEP 702 – Marking Deprecations Using the Type System | Peps.Python.Org." Accessed: Jul. 24, 2024. [Online]. Available: https://peps.python.org/pep-0702/

[56] E. Imgrund, "Locating and Editing Knowledge in Large Transformer Models for Code Generation." Jul. 2024.

[57] H. Touvron *et al.*, "Llama 2: Open Foundation and Fine-Tuned Chat Models." Accessed: Aug. 13, 2024. [Online]. Available: http://arxiv.org/abs/2307.09288

[58] A. Q. Jiang *et al.*, "Mistral 7B." Accessed: Aug. 13, 2024. [Online]. Available: http://arxiv.org/abs/2310.06825

[59] DeepSeek-AI *et al.*, "DeepSeek LLM: Scaling Open-Source Language Models with Longtermism." Accessed: Aug. 13, 2024. [Online]. Available: http://arxiv.org/abs/2401.02954

[60] J. Su, Y. Lu, S. Pan, A. Murtadha, B. Wen, and Y. Liu, "RoFormer: Enhanced Transformer with Rotary Position Embedding." Accessed: Aug. 13, 2024. [Online]. Available: http://arxiv.org/abs/2104.09864

[61] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai, "GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints." Accessed: Aug. 13, 2024. [Online]. Available: http://arxiv.org/abs/2305.13245

[62] B. Rozière *et al.*, "Code Llama: Open Foundation Models for Code." Accessed: Aug. 12, 2024. [Online]. Available: http://arxiv.org/abs/2308.12950

[63] "Beowolx/MistralHermes-CodePro-7B-v1 · Hugging Face." Accessed: Aug. 12, 2024. [Online]. Available: https://huggingface.co/beowolx/MistralHermes-CodePro-7B-v1

[64] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: Empowering Code Generation with OSS-Instruct." Accessed: Aug. 12, 2024. [Online]. Available: http://arxiv.org/abs/2312.02120

[65] M. Chen *et al.*, "Evaluating Large Language Models Trained on Code." Accessed: Jul. 07, 2024. [Online]. Available: http://arxiv.org/abs/2107.03374

[66] "@functools.Cache Destroys the Function Signature · Issue #11280 · Python/Typeshed." Accessed: Sep. 02, 2024. [Online]. Available: https://github.com/python/typeshed/issues/11280

[1] https://doi.org/10.1109/SANER.2016.39
[2] https://doi.org/10.1145/3520312.3534864
[3] https://doi.org/10.18653/v1/D18-2012

Listings

List of Figures

Figure 1: Generation strategies 13
Figure 2: Components of the IBMS 29
Figure 3: Format of a typical model generation result 30
Figure 4: Components of the LSP aided generation 33
Figure 5: The structure of the comment created by the “completion” interrupt type. The insert texts of all items are joined with “,”, similar to the colored bars. 61
Figure 6: The structure of the comment created by the “deprecation” interrupt type. For each item, a line with a hint is created, where the deprecation message is inserted at the position of the colored bar. 61
Figure 7: The structure of the comment created by the “signature” interrupt type. The green bar is replaced by the signature text of the item, while the pink text is replaced by the shortened documentation string. 61
Figure 8: Overview over the baseline evaluation. The first three models are averaged due to the use of sampling. The full stats are found in 80
Figure 9: Overview over the baseline evaluation using different model configurations. Copilot is not considered. The full results are found in 81
Figure 10: Overview over the baseline evaluation using different approach configurations. The full results are found in 81
Figure 11: Overview over the top 3 baseline configurations. The full results are found in 84
Figure 12: Average runtime in seconds per task, comparing the baseline and the approach. When comparing the median, the approach only less than doubles the runtime. When looking at the average runtime, it becomes clear that the approach incurs a larger penalty, sometimes tripling the runtime. 96
Figure 13: Possible language ids defined in the LSP specification 111

List of Tables

Table 1: Generation configuration parameters 13
Table 2: General requirements of the thesis 24
Table 3: Special requirements of the IBMS 28
Table 4: Special requirements of the dataset 36
Table 5: Dependencies in DependencyEval 40
Table 6: The experiments were run on the server Luke, with the specifications listed in the table. Only a single GPU is required for the experiments. 75
Table 7: Baseline evaluation results for each task. The table is sorted from best to worst solved tasks. Copilot results are included. 83
Table 8: Effects of using different models for evaluation of the approach, compared to the baseline. 84
Table 9: Effects of using sampling vs. not using sampling on the evaluation results, compared against the baseline. 85
Table 10: Effects of using different beam sizes on the evaluation results, compared against the baseline. 85
Table 11: Effects of using different approach configurations on the evaluation results, compared against the baseline. 85
Table 12: Difference between the baseline and the approach evaluation results for each task. The table is sorted from best to worst solved tasks. 86
Table 13: Overview over the top 3 configurations improving the baseline using the approach. 87
Table 14: All runtime errors occuring during the model generation when using the approach. The card displays the in total 17 errors, grouped into different categories, such as the models, model configurations, etc.. 91
Table 15: All evaluation errors occuring when using the approach, which did *not* occur in the baseline. The card displays the in total 80 errors, grouped into different categories, such as the models, model configurations, etc.. 93
Table 16: Size of dependencies, as counted using pygount. Documentation percentage is relative to the lines of code, thus excluding empty lines, etc. 94
Table 17: Overview over the baseline evaluation. The first three models are averaged due to the use of sampling. 125
Table 18: Overview over the baseline evaluation using different model configurations. Copilot is not considered. 125
Table 19: Overview over the baseline evaluation using different approach configurations. 125
Table 20: Overview over the top 3 baseline configurations. 125
Table 21: Copilot evaluation results 125

List of Code

Listing 1: Generating a sequence of tokens 12
Listing 2: Stopping criteria signature 14
Listing 3: Logits processor signature 14
Listing 4: CompletionItem structure 17
Listing 5: CompletionItemLabelDetails structure 17
Listing 6: MarkupContent structure 18
Listing 7: Completion message parameters 18
Listing 8: CompletionContext structure 18
Listing 9: CompletionList structure 18
Listing 10: SignatureHelp message parameters 19
Listing 11: SignatureHelp structure 19
Listing 12: SignatureInformation structure 19
Listing 13: ParameterInformation structure 19
Listing 14: Combination of fields to form the full solution in DependencyEval 39
Listing 15: Example code snippet for Pydantic showing the correct completion 41
Listing 16: Example code snippet for Textual showing the correct completion 42
Listing 17: The directory structure of the llm_lsp package. 44
Listing 18: The LspGenerationConfig data class, enabling customizable behavior. 45
Listing 19: The rearrange_according_to_beams method of the BeamTracker class, which reorders the items according to the sequence selections. 46
Listing 20: The BeamIndexStoringSearchScores class, which tracks the indices of selected sequences at each step and stores the final sequences before the final selection of the singular best sequence. 46
Listing 21: The InterruptStoppingCriteria class, which determines that the generation loop should be stopped when the interrupt token id occurs. 47
Listing 22: Entering the PyTorch device context manager before passing all arguments to the wrapped internal completion method. 49
Listing 23: The interrupt loop, which is the core of the Generator class and which enables the IBMS. Long argument lists are abbreviated by "..." 49
Listing 24: The format of the prompt used to predict the next symbol from the list of non deprecated completions directly. 52
Listing 25: The general structure employed for logging by the LogMixin. 53
Listing 26: The __call__ method of the LSPProcessor class. 54
Listing 27: Start of the scores_for_batch method, listing how the context retrieved from a language server. 55
Listing 28: The is_deprecated function, which determines if a completion item is deprecated. This function is called from a subprocess using the project environment. 57
Listing 29: The Interrupt dataclass, which is used to store the interrupt context and the interrupt token id. 58
Listing 30: The error message produced by PyTorch when the score of the interrupt token is set to the highest possible value. 59
Listing 31: A full sample prompt, including system and user prompt, comments and initial code. 60
Listing 32: The InterruptType abstract base class, which is used to implement the different interrupt types. 61

Listing 33: Waiting on to be returned completions from the language server. The asynchronous code is run in the synchronous code by retrieving the current outer event loop. 63
Listing 34: Directory structure of the dependency_eval repository. 65
Listing 35: The contents of the data/tasks/textual_1.py file, which is used to create the task textual_1 in the DependencyEval dataset. The file is split to obtain the imports, additional context, the function definition, function documentation string and gold standard completion. 66
Listing 36: The unit tests for the textual_1 task. The contents are stored in data/tests/textual_1.py. Next to the tested dependency, only the built-in unit test framework is used. The results are communicated via standard streams. 67
Listing 37: Typical console output when executing two Python unit tests. Parsing the output is error prone and unneeded, as the results can be accessed directly. 68
Listing 38: The general evaluation loop, which combines several each model configuration with a dataset task. 69
Listing 39: Sample output of the running evaluation. The first progress bar displays the progress in evaluation whole model configurations. The lower progress bar shows the progress of a single model configuration, showing the progress in evaluation a single task. 69
Listing 40: Arguments to Docker run, which starts the evaluation in a Docker container. Note that Git is installed to support installing dependencies from Git repositories. . . 71
Listing 41: The salt is a unique identifier appended to any name used during the evaluation. 71
Listing 42: Space used by the virtual environments in the cache. Results obtained using ncdu. Each directory is a Python virtual environment containing specific dependencies of a task. 72
Listing 43: Creating a virtual environment programmatically in Python. 73
Listing 44: Structure of the DependencyEval main file, which includes the CLI commands. 74
Listing 45: Base model configuration used for all models. 77
Listing 46: Example of the approach confusing the model to use a wrong value for an unneeded parameter. The approach guided the generation towards the correct method model_dump. 88
Listing 47: Example of the approach confusing the model to use a wrong type for a parameter value. The approach guided the generation towards the correct method text_area, but was confused by the Literal type hint. 88
Listing 48: Example of the approach confusing the model to use a wrong type hint for a return value. The approach guided the generation towards the correct method app, but was confused by the type hint. 89
Listing 49: Example of the approach confusing the model to use invalid parameters for a method. The approach guided the generation towards the correct method clear_meta_and_links, but was confused by the method signature. 89
Listing 50: Example of the correct code followed by an incorrect postfix, due to a bad format. 90
Listing 51: CompletionItemKind enum 112
Listing 52: Full JSON schema of the DependencyEval dataset. The schema defines the

structure of each task entry in the distributed JSONL file. 113
Listing 53: Task: bidict_1.py 114
Listing 54: Task: bidict_2.py 115
Listing 55: Task: dateutil_1.py 115
Listing 56: Task: dotted_1.py 115
Listing 57: Task: dotted_2.py 116
Listing 58: Task: emoji_1.py 116
Listing 59: Task: fastapi_1.py 117
Listing 60: Task: numpy_1.py 117
Listing 61: Task: pandas_1.py 118
Listing 62: Task: polars_1.py 118
Listing 63: Task: pydantic_1.py 119
Listing 64: Task: pydantic_2.py 119
Listing 65: Task: pydantic_3.py 120
Listing 66: Task: pytorch_1.py 120
Listing 67: Task: pytorch_2.py 120
Listing 68: Task: pytorch_3.py 121
Listing 69: Task: rich_1.py 121
Listing 70: Task: rich_2.py 121
Listing 71: Task: sklearn_1.py 122
Listing 72: Task: sklearn_2.py 122
Listing 73: Task: sqlalchemy_1.py 122
Listing 74: Task: textual_1.py 123
Listing 75: Task: textual_2.py 123
Listing 76: Task: theflow_1.py 124
Listing 77: Task: tsv2py_1.py 124
Listing 78: Signature of the model_dump method 127
Listing 79: Signature of the TextArea class 127
Listing 80: Signature of the clear_meta_and_links method 127
Listing 81: Copilot unable to correctly complete the pydantic_1 task, even with direct guiding comments in the prompt. The function call was completed by Copilot. 128

List of Examples

Example 1: Language model predicting the last word of the famous English pangram. 4
Example 2: Natural language encoded into tokens and token ids using a word-based vocabulary. 5
Example 3: Enforcing "yes" or "no" answers 8
Example 4: Few-shot prompting 9
Example 5: Context injection 9
Example 6: Typical execution of Python unit tests through the command line. 68

Appendix

A Language Server Protocol

Language	Language ID
ABAP	abap
Windows Bat	bat
BibTeX	bibtex
Clojure	clojure
Coffeescript	coffeescript
C	c
C++	cpp
C#	csharp
CSS	css
Diff	diff
Dart	dart
Dockerfile	dockerfile
Elixir	elixir
Erland	erlang
F#	fsharp
Git	git-commit and git-rebase
Go	go
Groovy	groovy
Handlebars	handlebars
HTML	html
Ini	ini
Java	java
JavaScript	javascript
JavaScript React	javascriptreact
JSON	json
LaTeX	latex
Less	less
Lua	lua
Makefile	makefile
Markdown	markdown
Objective-C	objective-c
Objective-C++	objective-cpp
Perl	perl
Perl 6	perl6
PHP	php
Powershell	powershell
Pug	jade
Python	python
R	r
Razor	razor
Ruby	ruby
Rust	rust
SCSS	scss or sass
Scala	scala
ShaderLab	shaderlab
Shellscript	shellscript
SQL	sql
Swift	swift
TypeScript	typescript
TypeScript React	typescriptreact
TeX	tex
Visual Basic	vb
XML	xml
XSL	xsl
YAML	yaml

Figure 13: Possible language ids defined in the LSP specification

```typescript
namespace CompletionItemKind {
    const Text: 1;
    const Method: 2;
    const Function: 3;
    const Constructor: 4;
    const Field: 5;
    const Variable: 6;
    const Class: 7;
    const Interface: 8;
    const Module: 9;
    const Property: 10;
    const Unit: 11;
    const Value: 12;
    const Enum: 13;
    const Keyword: 14;
    const Snippet: 15;
    const Color: 16;
    const File: 17;
    const Reference: 18;
    const Folder: 19;
    const EnumMember: 20;
    const Constant: 21;
    const Struct: 22;
    const Event: 23;
    const Operator: 24;
    const TypeParameter: 25;
}
type CompletionItemKind = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25;
```

Listing 51: CompletionItemKind enum

B DependencyEval Dataset

Schema

JSON Schema

```
{
  "$schema": "https://json-schema.org/draft/2020-12/json-schema-core",
  "type": "object",
  "properties": {
    "task_id": {
      "type": "string",
      "pattern": "^PackageEval_\\d+$"
    },
    "task_name": {
      "type": "string",
      "pattern": "^.*_\\d+$"
    },
    "test_code": {
      "type": "string"
    },
    "import_statements": {
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "package_dependencies": {
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "context": {
      "type": "string"
    },
    "function_signature": {
      "type": "string"
    },
    "function_documentation": {
      "type": "string"
    },
    "entry_point": {
      "type": "string"
    },
    "solution": {
      "type": "string"
    },
    "reason": {
      "type": "string"
    },
    "kind": {
      "type": "string"
    },
    "date": {
      "type": "string",
      "format": "date"
    },
    "code_kind": {
```

```
      "type": "string"
    },
    "modification_kind": {
      "type": "string"
    },
    "changelog": {
      "type": "string",
      "format": "uri"
    },
    "python_version": {
      "type": "string"
    },
    "package_name": {
      "type": "string"
    }
  },
  "required": [
    "task_id",
    "task_name",
    "test_code",
    "import_statements",
    "package_dependencies",
    "context",
    "function_signature",
    "function_documentation",
    "entry_point",
    "solution",
    "reason",
    "kind",
    "date",
    "code_kind",
    "modification_kind",
    "changelog",
    "python_version"
  ]
}
```

Listing 52: Full JSON schema of the DependencyEval dataset. The schema defines the structure of each task entry in the distributed JSONL file.

Tasks

```python
from typing import Any, Dict

from bidict import OnDup, OnDupAction, bidict


def insert_values_drop_old_on_dup(values: bidict, items: Dict[str, Any]):
    """Insert all key value pairs from items into values at once. Drop old keys and values on duplication.

    Args:
        values (bidict): Bidirectional mapping between keys and values
        items (Dict[str, Any]): Python mapping between keys and values to be inserted into values
    """
    values.putall(items, OnDup(key=OnDupAction.DROP_OLD, val=OnDupAction.DROP_OLD))
```

Listing 53: Task: bidict_1.py

```python
from bidict import bidict


def invert_bidict_direction(values: bidict) -> bidict:
    """Return the inverse of the given bidirectional mapping instance.

    Args:
        values (bidict): Bidirectional mapping between any keys and values

    Returns:
        bidict: Inverse of values
    """
    return values.inverse
```

Listing 54: Task: bidict_2.py

```python
from datetime import datetime

import dateutil


def current_datetime_in_local_timezone() -> datetime:
    """Return the current date and time in the local time zone.

    Returns:
        datetime: Current local date and time
    """
    return datetime.now(dateutil.tz.tzlocal())
```

Listing 55: Task: dateutil_1.py

```python
from dotted.collection import DottedDict


def get_user_street_name(user: DottedDict) -> str:
    """Retrieve the street name of the user.

    Args:
        user (DottedDict): The user has the following JSON schema:
    {
        name: str,
        age: int,
        email: str,
        street: {
            number: int,
            name: str
        }
    }

    Returns:
        str: Street name
    """
    return user["street.name"]
```

Listing 56: Task: dotted_1.py

```python
from typing import Any

from dotted.collection import DottedList


def get_2d_board_entry(board: DottedList, index: str) -> Any:
    """Retrieve the value in the 2d board at the given index.

    Args:
        board (DottedList): A 2d dimensional board implemented through nested lists. The board may store any type of value.
        index (str): An index in the format of "column.row"

    Returns:
        Any: The value
    """
    return board[index]
```

Listing 57: Task: dotted_2.py

```python
import emoji

THUMBS_UP = emoji.emojize(":thumbs_up:")
THUMBS_DOWN = emoji.emojize(":thumbs_down:")

def does_the_text_contain_only_emojis(text: str) -> str:
    """Return either thumbs up or down depending on text containing only emojis.

    Args:
        text (str): Any input text

    Returns:
        str: Thumbs up emoji if text only contains emojis. Else thumbs down.
    """
    return THUMBS_UP if emoji.purely_emoji(text) else THUMBS_DOWN
```

Listing 58: Task: emoji_1.py

```python
from contextlib import asynccontextmanager

from fastapi import FastAPI


def startup(app: FastAPI):
    app.state.startup = True

def shutdown(app: FastAPI):
    app.state.shutdown = True

def create_fastapi_app() -> FastAPI:
    """Create a new FastAPI app which calls the lifespan functions startup and shutdown.

    Returns:
        FastAPI: New FastAPI instance
    """
    @asynccontextmanager
    async def lifespan(app: FastAPI):
        startup(app)
        yield
        shutdown(app)
    return FastAPI(lifespan=lifespan)
```

Listing 59: Task: fastapi_1.py

```python
import numpy as np
from typing import List

def add_strings_element_wise(a: List[str], b: List[str]) -> List[str]:
    """Add both lists of strings element-wise. Use the Numpy library.

    Args:
        a (List[str]): First list
        b (List[str]): Second list

    Returns:
        List[str]: Combined list
    """
    return np.strings.add(a, b)
```

Listing 60: Task: numpy_1.py

```python
import pandas as pd

def get_first_group_entry_allow_na(grouped_df: pd.core.groupby.GroupBy) -> pd.core.generic.NDFrameT:
    """Return the first row for each group, while not skipping NA values.

    Args:
        grouped_df (pd.core.groupby.GroupBy): The already grouped data frame.

    Returns:
        pd.core.generic.NDFrameT: A generic multi dimensional dataframe, containing each first row result.
    """
    return grouped_df.first(skipna=False)
```

Listing 61: Task: pandas_1.py

```python
from typing import List

import polars as pl


def lazy_filter_old_users(csv_file_path: str) -> List[str]:
    """Lazily return a list of all user names, which are older than 50. The name column is `name`, the age column is `age`.

    Args:
        csv_file_path (str): Path to a CSV file with input data

    Returns:
        List[str]: User names of users older than 50
    """
    df = pl.scan_csv(csv_file_path)
    names = df.filter(pl.col("age") > 50).select(pl.col("name")).collect()
    return names.to_series().to_list()
```

Listing 62: Task: polars_1.py

```python
from typing import Any, Dict

from pydantic import BaseModel


class User(BaseModel):
    name: str
    email: str
    age: int

def convert_user_to_dict(user: User) -> Dict[str, Any]:
    """Convert the user into a Python dictionary.

    Args:
        user (User): Pydantic user model

    Returns:
        Dict[str, Any]: User attributes as a Python key value mapping
    """
    return user.model_dump()
```

Listing 63: Task: pydantic_1.py

```python
from typing import Any, Dict

from pydantic import BaseModel


class User(BaseModel):
    name: str
    email: str
    age: int

def convert_user_to_json(user: User) -> str:
    """Convert the given user model into a JSON string.

    Args:
        user (User): Pydantic user model

    Returns:
        str: JSON string of user attributes
    """
    return user.model_dump_json()
```

Listing 64: Task: pydantic_2.py

```python
from typing import Any, Dict

from pydantic import BaseModel


class User(BaseModel):
    name: str
    email: str
    age: int

def duplicate_user(user: User) -> User:
    """Duplicate the user.

    Args:
        user (User): The Pydantic user model

    Returns:
        User: Deep copy of the user
    """
    return user.model_copy()
```

Listing 65: Task: pydantic_3.py

```python
from torch.nn import CrossEntropyLoss


def create_sum_cross_entropy_loss_module() -> CrossEntropyLoss:
    """Create an instance of CrossEntropyLoss which computes the sum of the cross entropy loss.

    Returns:
        CrossEntropyLoss: New instance which computes the sum of the cross entropy loss
    """
    return CrossEntropyLoss(reduction="sum")
```

Listing 66: Task: pytorch_1.py

```python
from numbers import Number

import torch


def create_1d_tensor_in_range(start: Number, end: Number) -> torch.Tensor:
    """Return a 1d tensor with values from start to end.

    Args:
        start (Number): Starting number (inclusive)
        end (Number): End number (exclusive)

    Returns:
        torch.Tensor: Tensor with values from start to end
    """
    return torch.arange(start, end)
```

Listing 67: Task: pytorch_2.py

```python
import torch


def calculate_cholesky(input: torch.Tensor) -> torch.Tensor:
    """Calculate the Cholesky decomposition.

    Args:
        input (torch.Tensor): Input tensor

    Returns:
        torch.Tensor: Cholesky decomposition of input tensor
    """
    return torch.linalg.cholesky(input)
```

Listing 68: Task: pytorch_3.py

```python
from rich.style import Style


def clear_style(style: Style) -> Style:
    """Obtain a copy of style with all meta and links removed.

    Args:
        style (Style): target style

    Returns:
        Style: target style without meta and links
    """
    return style.clear_meta_and_links()
```

Listing 69: Task: rich_1.py

```python
from rich.prompt import Prompt


def create_case_insensitive_prompt(text: str) -> Prompt:
    """Create a prompt instance, providing the text and using case insensitivity.

    Args:
        text (str): prompt text

    Returns:
        Prompt: created prompt
    """
    return Prompt(text, case_sensitive=False)
```

Listing 70: Task: rich_2.py

```python
from sklearn.preprocessing import OneHotEncoder


def create_dense_one_hot_encoder() -> OneHotEncoder:
    """Create a OneHotEncoder which encodes categorical features into a dense matrix.

    Returns:
        OneHotEncoder: New instance of OneHotEncoder encoding categorical features into a dense matrix
    """
    return OneHotEncoder(sparse_output=False)
```

Listing 71: Task: sklearn_1.py

```python
from sklearn.preprocessing import OneHotEncoder


def create_polars_compatible_one_hot_encoder() -> OneHotEncoder:
    """Create a OneHotEncoder which encodes categorical features into polars containers.

    Returns:
        OneHotEncoder: New instance of OneHotEncoder encoding categorical features into polars containers
    """
    encoder = OneHotEncoder()
    encoder.set_output(transform="polars")
    return encoder
```

Listing 72: Task: sklearn_2.py

```python
from sqlalchemy import Row
from sqlalchemy.engine.row import _TP


def get_tuple_of_row(row: Row) -> _TP:
    """Return this row as a tuple.

    Args:
        row (Row): Input row

    Returns:
        _TP: Input row represented as a tuple
    """
    return row._tuple()
```

Listing 73: Task: sqlalchemy_1.py

```python
from textual.widgets import TextArea


def create_textual_text_area_with_indent() -> TextArea:
    """Create a TextArea widget, which indents its content when tab is pressed.

    Returns:
        TextArea: New instance of TextArea with indentation on tab press
    """
    return TextArea(tab_behavior="indent")
```

Listing 74: Task: textual_1.py

```python
from textual.app import App
from textual.types import AnimationLevel


def create_app_without_animations() -> App:
    """Create a minimal textual App without animations.

    Returns:
        App: New App instance with disabled animations
    """
    app = App()
    app.animation_level = "none"
    return app
```

Listing 75: Task: textual_2.py

```python
from theflow import Function


def square(x: int) -> int:
    return x*x

class MultiplyBy(Function):
    factor: int
    def run(self, y):
        return y*self.factor



class MultiplySquareFlow(Function):
    multiply: Function
    square: Function

    def run(self, x):
        y = self.multiply(x)
        y = self.square(y)
        return y

def multiply_then_square(x: int, multiplication_factor: int) -> int:
    """Multiply x by multiplication factor, then square the result.

    Args:
        x (int): Input number
        multiplication_factor (int): Multiplication factor for x

    Returns:
        int: x times multiplication factor, then the squared result using the provided Functions
    """
    flow = MultiplySquareFlow(square=square,multiply=MultiplyBy(factor=multiplication_factor))
    return flow(x=x)
```

Listing 76: Task: theflow_1.py

```python
from datetime import datetime
from typing import Any, List, Tuple

from tsv.helper import Parser


def parse_tsv_file(filename: str) -> List[Tuple[Any, ...]]:
    """The file at filepath contains entries in the tsv format. Parse the file into a Python list of tuples.

    Args:
        filename (str): Name of the TSV file. The TSV entries have the following columns: name, age, birthday

    Returns:
        List[Tuple[Any, ...]]: List of Python tuples of the tabular data
    """
    parser = Parser(fields=(str, int, datetime))
    with open(filename, "rb") as f:
        return parser.parse_file(f)
```

Listing 77: Task: tsv2py_1.py

C Evaluation Result Tables

Table 17: Overview over the baseline evaluation. The first three models are averaged due to the use of sampling.

Name	Fully solved	Partially solved	Not solved	Error
CodeLlama	6.67%	36.89%	42%	14.44%
Magicoder	7.33%	29.11%	22.22%	41.33%
MistralHermes	4%	19.56%	26.89%	49.56%
Copilot	8%	44%	36%	12%
Averaged w.o. Copilot	6%	28.52%	30.37%	35.11%
Averaged	6.04%	28.8%	30.47%	34.69%

Table 18: Overview over the baseline evaluation using different model configurations. Copilot is not considered.

Name	Fully solved	Partially solved	Not solved	Error
Sampling	7.26%	26.52%	29.63%	36.59%
No Sampling	4.74%	30.52%	31.11%	33.63%
1 Beam	0%	10.67%	22.67%	66.67%
2 Beams	7.78%	38.22%	35.78%	18.22%
3 Beams	10.22%	36.67%	32.67%	20.44%

Table 19: Overview over the baseline evaluation using different approach configurations.

Name	Fully solved	Partially solved	Not solved	Error
Default	5.78%	28.67%	30%	35.56%
Masking	6.22%	28.22%	29.11%	36.44%
Use Completions	6%	28.67%	32%	33.33%

Table 20: Overview over the top 3 baseline configurations.

Name	Fully solved	Partially solved	Not solved	Error
Masking Magicoder sampling 2 beams	20%	44%	28%	8%
Default Magicoder sampling 3 beams	16%	32%	40%	12%
Masking MistralHermes sampling 3 beams	16%	28%	32%	24%

Table 21: Copilot evaluation results

Fully solved	Partially solved	Not solved	Errors
dateutil_1	bidict_1	dotted_1	sklearn_1
pytorch_1	bidict_2	fastapi_1	textual_1
	dotted_2	pandas_1	theflow_1
	emoji_1	polars_1	
	numpy_1	rich_1	
	pydantic_1	rich_2	
	pydantic_2	sqlalchemy_1	
	pydantic_3	textual_2	
	pytorch_2	tsv2py_1	
	pytorch_3		
	sklearn_2		

D Evaluation Result Details

The following code snippets contain signature help information on different functions used in tasks from the DependencyEval dataset. The Pydantic model_dump method has the following signature:

```python
def model_dump(
    *,
    mode: Literal["json", "python"] | str = "python",
    include: IncEx = None,
    exclude: IncEx = None,
    context: Any | None = None,
    by_alias: bool = False,
    exclude_unset: bool = False,
    exclude_defaults: bool = False,
    exclude_none: bool = False,
    round_trip: bool = False,
    warnings: (
        bool | Literal["none", "warn", "error"]
    ) = True,
    serialize_as_any: bool = False
) -> dict[str, Any]
```

Listing 78: Signature of the model_dump method

The TextArea class of the rich library has the following signature:

```python
class TextArea(
        text: str="",
        *,
        language: str | None=None,
        theme: str="css",
        soft_wrap: bool=True,
        tab_behavior: Literal["focus", "indent"]="focus",
        read_only: bool=False,
        show_line_numbers: bool=False,
        line_number_start: int=1,
        max_checkpoints: int=50,
        name: str | None=None,
        id: str | None=None,
        classes: str | None=None,
        disabled: bool=False,
        tooltip: RenderableType | None=None
)
```

Listing 79: Signature of the TextArea class

The method clear_meta_and_links of the Style class in the rich library has the following signature:

```python
instance _lru_cache_wrapper(*args: Hashable, **kwargs: Hashable) -> _T
```

Listing 80: Signature of the clear_meta_and_links method

As seen, the signature is not as expected, as the method is decorated with @lru_cache, which destroys the original signature [66].

The code block Listing 81 displays how Copilot fails to complete the pydantic_1 task, even when the prompt contains guiding comments.

```python
from typing import Any, Dict

from pydantic import BaseModel


class User(BaseModel):
    name: str
    email: str
    age: int


def convert_user_to_dict(user: User) -> Dict[str, Any]:
    """Convert the user into a Python dictionary.

    Args:
        user (User): Pydantic user model

    Returns:
        Dict[str, Any]: User attributes as a Python key value mapping
    """
    # the function user.dict is deprecated, and user.model_dump should be used instead
    return user.dict()
```

Listing 81: Copilot unable to correctly complete the pydantic_1 task, even with direct guiding comments in the prompt. The function call was completed by Copilot.

QR code

Scan the QR code to access the source code of this thesis:
https://github.com/data-niklas/master_thesis