

# Black-box context-aware code completion

Enhancing consumer-facing code completion with  
low-cost general enhancements

Tim van Dam

# Black-box context-aware code completion

Enhancing consumer-facing code completion  
with low-cost general enhancements

by

Tim van Dam

to obtain the degree of Master of Science  
at the Delft University of Technology,  
to be defended on Monday July 8, 2024 at 8:00 AM.

Student number: 5086027  
Project duration: November 13, 2023 – July 8, 2024  
Thesis committee: Prof. dr. A. van Deursen, TU Delft, chair  
Assistant Prof. dr. M. Izadi, TU Delft, supervisor  
Assistant Prof. dr. J. Yang, TU Delft

Cover: Nieuwe Kerk, Delft by <https://unsplash.com/@thezenoeffect>  
Style: TU Delft Report Style, with modifications by Daan Zwaneveld

An electronic version of this thesis is available at <http://repository.tudelft.nl/>.

# Preface

I have been working on LLMs for code and AI for Software Engineering (AI4SE) since my bachelor thesis, where I analyzed what types of contextual information leads to the best code completion performance. While I was not fully convinced it was my thing at first, I have definitely grown into it, driving me to continue on the path of researching LLMs for code. This has led to me writing this thesis, where I explore whether it is possible to improve LLM-based code completion by having an n-gram model provide context from the current project. I am happy with the result and hope to have made a small dent in the field.

## Acknowledgements

I would not have been able to complete this thesis without the support and encouragement of many people.

First and foremost, I want to extend my heartfelt thanks to my daily supervisor, Dr. Maliheh (Mali) Izadi, and my thesis advisor, Prof. Dr. Arie van Deursen. Your unwavering support, insight, and encouragement throughout this thesis and other academic ventures during my studies have been instrumental in my success and growth.

I also wish to express my gratitude to Egor Bogomolov for his invaluable support and direction throughout the course of this thesis. Your ideas and observations have helped me shape the direction of this work.

I also sincerely thank everyone at JetBrains. Your support, hospitality, and endless supply of delicious coffee made the journey much more enjoyable.

Furthermore, I am grateful to the AISE Lab for the knowledge and companionship during the weekly reading clubs. The discussions and insights shared have been invaluable.

Finally, I extend my deepest appreciation to my parents and brothers for their unwavering support and encouragement at every step of this thesis and throughout my life. Your belief in me has been a constant source of strength and motivation.

*Tim van Dam  
Delft, July 2024*

# Summary

Code completion is a common tool in software engineering that has greatly improved as a result of advancements in large language models. Despite these advancements, code completion models often struggle in new environments like private repositories that are not present in the training data of these models. While solutions like retrieval-augmented generation and fine-tuning can improve accuracy in such situations, they are generally impractical for consumer products due to their computational overhead.

This thesis explores whether a hybrid approach that combines a global large language model with a local n-gram model can improve the accuracy of code completion in new environments. The global large language model is responsible for writing syntactically correct code, while the local n-gram model guides predictions towards contextually correct solutions. Building on previous work by Tu, Su, and Devanbu, this thesis experiments and analyses approaches based on Bayesian inference, interpolation, dynamic weighting, masking, and conditional application.

We demonstrate that modern language models are able to capture localness from in-file context more effectively than n-gram models can based on full project context. Our findings reveal that targeting parts of code that are highly sensitive to project context, such as identifiers, can lead to improved performance. We show that combining masking techniques with Bayesian inference results in the best overall performance, and that LLMs lack awareness of their limitations in new contexts.

While absolute performance gains are limited, the low overhead of our approach makes it a viable option to integrate similar approaches inside of consumer hardware in real-world scenarios. This integration could improve code completion performance at a low cost, albeit by a slight margin.

In conclusion, this thesis rigorously investigates global-local hybrid models, demonstrating the effectiveness of large language models compared to n-gram models, and establishing techniques for improving code completion accuracy in new contexts.

# Contents

<b>Preface</b>	<b>i</b>
<b>Summary</b>	<b>ii</b>
<b>Nomenclature</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background and Related Work</b>	<b>3</b>
2.1 Large Language Models . . . . .	3
2.1.1 Tokenization . . . . .	3
2.1.2 The Transformer Architecture . . . . .	4
2.1.3 N-gram Models . . . . .	4
2.2 White-box Generation Techniques . . . . .	5
2.2.1 Retrieval-Augmented Generation . . . . .	5
2.2.2 Downsides of White-Box Enhancements . . . . .	5
2.3 Black-Box Generation Techniques . . . . .	5
2.4 Related Work . . . . .	5
2.4.1 Retrieval-Augmented Generation . . . . .	6
2.4.2 Adaptive Code Completion with White-Box Retrieval . . . . .	6
<b>3 Research Objectives and Approach</b>	<b>7</b>
<b>4 Experimental Setup</b>	<b>9</b>
4.1 Dataset . . . . .	9
4.2 Model . . . . .	9
4.3 Metrics . . . . .	10
4.4 Architecture . . . . .	10
<b>5 Is Localness all You Need?</b>	<b>12</b>
5.1 Exploring LLM + N-gram Bayesian inference . . . . .	12
5.1.1 Training . . . . .	13
5.1.2 Dataset Creation . . . . .	13
5.1.3 Bayesian Inference . . . . .	13
5.1.4 Dynamic weighting . . . . .	14
5.1.5 Interpolation and Smoothing . . . . .	15
5.1.6 Shortcomings of N-gram Models . . . . .	19
<b>6 Targeted Identifier Completion</b>	<b>21</b>
6.1 Shortcomings of LLMs . . . . .	21
6.1.1 Applicability of N-gram Models . . . . .	21
6.2 Targeted LLM steering . . . . .	23
6.2.1 Identifiers and Function Invocations . . . . .	23
6.2.2 Targeted Bayesian Inference . . . . .	24
6.2.3 N-gram Masking Techniques . . . . .	28
6.2.4 LLM Retrievers . . . . .	32
6.2.5 N-Gram-LLM Retriever-Ranker System . . . . .	36
6.2.6 Conditional Application . . . . .	40
<b>7 Run-Time Efficiency</b>	<b>43</b>
<b>8 Discussion</b>	<b>45</b>
8.1 RQ1: LLM + N-gram Bayesian Inference . . . . .	45

---

8.2	RQ2: Parts of Code Dependent on Project Context . . . . .	45
8.3	RQ3: Targeted Identifier Completion . . . . .	45
8.3.1	RQ3.1: Masking Techniques . . . . .	45
8.3.2	RQ3.2: Masking Components and Project Context . . . . .	46
8.3.3	RQ3.3: Masking with Bayesian Inference . . . . .	46
8.4	RQ4: Conditional Application . . . . .	46
8.5	RQ5: TIC Overhead . . . . .	46
<b>9</b>	<b>Conclusions</b>	<b>47</b>
<b>10</b>	<b>Future Work</b>	<b>48</b>
10.1	Steering with Language Features . . . . .	48
10.2	Human-in-the-loop . . . . .	48
10.3	Evaluating Real-World Scenarios . . . . .	48
10.4	Local Light-Weight Multi-Layer Perceptrons . . . . .	48
10.5	Different Programming Languages . . . . .	49
	<b>References</b>	<b>50</b>
<b>A</b>	<b>Architecture UML Class Diagram</b>	<b>53</b>
<b>B</b>	<b>Mid-Token Invocation</b>	<b>55</b>



# List of Figures

5.1	Regular Expression for determining the Cursor Position in Line Code Completion (\s matches whitespace characters, \S matches non-whitespace characters)	13
5.2	Next-token Accuracy for BI Hybrid Model	14
5.3	DBI Parameter Values Weights	15
5.4	DBI Hybrid Model Next-token Accuracy	16
5.5	BI Hybrid Model Next-Token Accuracy (Interpolation and Smoothing)	18
5.6	DBI Hybrid Model Next-Token Accuracy (Interpolation and Smoothing)	18
6.1	Regular expression for identifying Identifiers	22
6.2	Function Invocation Grammar	22
6.3	Regular expression for tokenizing Code	23
6.4	Identifier Tokens (Top) versus LLM Tokenizer Tokens (Bottom)	23
6.5	BI Hybrid Model Next-token Accuracy (Identifiers)	25
6.6	DBI Hybrid Model Next-token Accuracy (Identifiers)	26
6.7	BI Hybrid Model Next-token Accuracy (Function Invocations)	27
6.8	DBI Hybrid Model Next-token Accuracy (Function Invocations)	27
6.9	Top-k Masking Hybrid Model Next-token Accuracy (Identifiers)	29
6.10	Top-k Masking Precision and Recall (Identifiers)	29
6.11	Top-p Masking Hybrid Model Next-token Accuracy (Identifiers)	30
6.12	Top-p Masking Average Number of Tokens (Identifiers)	31
6.13	Top-p Masking Precision and Recall (Identifiers)	31
6.14	Top-k Masking Hybrid Model Next-token Accuracy (Identifiers, LLM Retriever)	33
6.15	Top-k Masking Precision and Recall (Identifiers, LLM Retriever)	33
6.16	Top-p Masking Hybrid Model Next-token Accuracy (Identifiers, LLM Retriever)	34
6.17	Top-p Masking Precision and Recall (Identifiers, LLM Retriever)	34
6.18	Top-p Masking Average Number of Tokens (Identifiers, LLM Retriever)	35
6.19	Tie-Breaking Masking Hybrid Model Next-token Accuracy (Identifiers, LLM Retriever)	35
6.20	Tie-Breaking Masking Precision and Recall (Identifiers, LLM Retriever)	36
6.21	Tie-Breaking Masking Average Number of Tokens (Identifiers, LLM Retriever)	36
6.22	Top-k Masking + BI Hybrid Model Next-token Accuracy (Identifiers)	38
6.23	Top-p Masking + BI Hybrid Model Next-token Accuracy (Identifiers)	39
6.24	Top-1 Threshold Activation Receiver Operating Characteristic (ROC) Curve	41
6.25	Top-2D Threshold Activation Receiver Operating Characteristic (ROC) Curve	41
6.26	In-n-gram Threshold Activation Receiver Operating Characteristic (ROC) Curve	42
7.1	Benchmark Results	44
A.1	UML Diagram of the Code Architecture	54

# List of Tables

5.1	BI Hybrid Model Accuracy . . . . .	14
5.2	DBI Hybrid Model Accuracy . . . . .	15
5.3	BI and DBI Hybrid Model with Interpolation and Smoothing Accuracy . . . . .	17
6.1	BI and DBI Hybrid Model Accuracy (Identifiers) . . . . .	24
6.2	BI and DBI Hybrid Model Accuracy (Function Invocations) . . . . .	26
6.3	Top-k/p Masking Hybrid Model Accuracy (Identifiers) . . . . .	28
6.4	Top-k/p/Tie-Breaking Masking Hybrid Model Accuracy (Identifiers, LLM Retriever) . . . .	32
6.5	Top-k/p Masking + BI Hybrid Model Accuracy (Identifiers) . . . . .	37
6.6	Top-k Masking + Activation Criterion Accuracy Hybrid Model Accuracy (Identifiers) . . . .	40
B.1	Mid-Token Invocation Next-Token Accuracy Results (Izadi et al.) . . . . .	56
B.2	Mid-Token Invocation Next-Token Accuracy Results (Unseen Dataset) . . . . .	56



# List of Experiments

5.1	Baseline LLM Code Completion . . . . .	13
5.2	UB on BI Accuracy . . . . .	14
5.3	BI Accuracy with $w_{ngram} = 0.4$ . . . . .	14
5.4	UB on DBI Accuracy . . . . .	14
5.5	DBI Accuracy with $\gamma = 71$ . . . . .	15
5.6	UB on BI Accuracy with Interpolation and Smoothing . . . . .	17
5.7	BI Accuracy with Interpolation and Smoothing using $w_{ngram} = 0.4$ . . . . .	19
5.8	DBI Accuracy with Interpolation and Smoothing using $\gamma = 71$ . . . . .	19
6.1	Baseline LLM Code Completion (Identifiers) . . . . .	24
6.2	UB on BI Accuracy (Identifiers) . . . . .	25
6.3	BI Accuracy with $w_{ngram} = 0.5$ (Identifiers) . . . . .	25
6.4	UB on DBI Accuracy (Identifiers) . . . . .	25
6.5	DBI Accuracy with $\gamma = 51$ (Identifiers) . . . . .	25
6.6	Baseline LLM Code Completion (Function Invocations) . . . . .	26
6.7	UB on BI Accuracy (Function Invocations) . . . . .	26
6.8	UB on DBI Accuracy (Function Invocations) . . . . .	27
6.9	UB on Top-k Masking Accuracy (Identifiers) . . . . .	28
6.10	Top-k Masking Accuracy with $k = 19$ (Identifiers) . . . . .	30
6.11	UB on Top-p Masking Accuracy (Identifiers) . . . . .	30
6.12	Top-p Masking Accuracy with $p = 0.9$ (Identifiers) . . . . .	31
6.13	UB on Top-k Masking Accuracy (Identifiers, LLM Retriever) . . . . .	32
6.14	UB on Top-p Masking Accuracy (Identifiers, LLM Retriever) . . . . .	33
6.15	UB on Tie-Breaking Masking Accuracy (Identifiers, LLM Retriever) . . . . .	35
6.16	UB on Top-k Masking + Bayesian Inference Accuracy (Identifiers) . . . . .	37
6.17	Top-k Masking + BI Accuracy with $w_{ngram} = 0.5$ and $k = 19$ (Identifiers) . . . . .	37
6.18	UB on Top-p Masking + BI Accuracy (Identifiers) . . . . .	37
6.19	Top-p Masking + BI Accuracy with $w_{ngram} = 0.5$ and $p = 0.9$ (Identifiers) . . . . .	37
6.20	UB on Top-k Masking Accuracy with Top-1 Threshold Activation (Identifiers) . . . . .	40
6.21	UB on Top-2D Masking Accuracy with Top-1 Threshold Activation (Identifiers) . . . . .	41
6.22	UB on Top-k Masking Accuracy with In-n-gram Threshold Activation (Identifiers) . . . . .	42
6.23	Top-k Masking Accuracy with In-n-gram Threshold Activation with $k = 19$ (Identifiers) . . . . .	42
B.1	Mid-Token Invocation Pre-Post Processing – Fibonacci . . . . .	55
B.2	Mid-Token Invocation Pre-Post Processing – Izadi et al. . . . .	55
B.3	Mid-Token Invocation Pre-Post Processing – Unseen Dataset . . . . .	56

# Nomenclature

## Abbreviations

Abbreviation	Definition
AI	Artificial Intelligence
BI	Bayesian Inference
DBI	Dynamic Bayesian Inference
AST	Abstract Syntax Tree
AUC	Area Under Curve
BLEU	Bilingual Evaluation Understudy
BPE	Byte-Pair Encoding
CI	Continuous Integration
EOS	End of Sequence
FPR	False Positive Rate
GPT	Generative Pre-trained Transformer
GRU	Gated Recurrent Unit
IDE	Integrated Development Environment
IR	Information Retrieval
LLM	Large Language Model
LM	Language Model
LSP	Language Server Protocol
LSTM	Long Short-Term Memory
MLP	Multi-Layer Perceptron
NLP	Natural Language Processing
NN	Neural Network
OOD	Out of Distribution
OOV	Out of Vocabulary
RAG	Retrieval Augmented Generation
RNN	Recurrent Neural Network
ROC	Receiver Operating Characteristic
ROUGE	Recall-Oriented Understudy for Gisting Evaluation
TIC	Targeted Identifier Completion
TPR	True Positive Rate
UB	Upper Bound
UML	Unified Modeling Language

# 1

## Introduction

Code completion has long been a staple in any developer's toolkit. Recent advancements in neural code completion have lead to transformer-based code completion models becoming the de facto standard, with models like Codex and other GPT-based models like StarCoder and CodeGen providing accurate suggestions at low latencies [2, 3, 4, 5, 6, 7, 8, 9, 10]. Traditional rule-based code completion systems, typically built into integrated development environments (IDEs) can intelligently suggest variables, methods, and more based on static analysis of the codebase. Transformer-based code completion systems work differently, treating code as natural language and attempt code completion through language modeling rather than analyzing a code as a tree. This new approach has rapidly gained popularity due to its versatile and accurate nature. Rule-based code completion systems differ in this aspect, and are constrained by the information that can be inferred exactly from the codebase. This leads to precise suggestions within a narrow scope. As an example, a common rule infers which variables fit as an argument to a function by matching parameter types with variable types. Yet since the majority of coding is not nearly as exact, this leads to limited applicability compared to transformer-based code completion models. Relying on language modeling expands the scope of code completion immensely by shifting the focus away from constructing abstract syntax trees (ASTs) that meet some constraints. Instead, language modeling focuses on writing syntactically valid code rather than generating code that is very likely to be valid. This broadened scope leads to models that can write large sequences of syntactically valid code, which many developers find to be more productive than rule-based systems that will suggest symbols one at a time.

While language modeling has brought significant improvements to productivity, it also suffers from a lack of precision in new environments. Rule-based systems can traverse the file system to find relevant classes, type declarations, and methods, making them perform well in any environment. Language models simply do not work this way: no environment information except for the information present in the input is used by these models, leading to poor performance in out of distribution (OOD) environments. Such environments are not rare: common examples are private repositories, likewise, environments that use versions of libraries that are different from the versions present in a language model's training data also encounter such issues. Many approaches can resolve these issues, notably retrieval-augmented generation (RAG) and fine-tuning. RAG involves retrieving relevant documents and using these as supplementary input to the language model, whether it be as a vector representation or as natural language affixed to the user input [11, 12, 13]. This relatively simple approach can be used to ground the model to the current project by providing information on relevant available functions, classes, and more. While this is generally a valid approach, it does have flaws. First off, RAG relies on the availability of an accurate retrieval system, which not only adds overhead, but also adds complexity as collecting data and defining what exactly is relevant is not a trivial task. Next, affixing to prompts can lead to performance degradation. While modern language models have large enough input sizes, they often are not able to attend to every part of the input equally [14]. Models are especially biased towards the start and the end of inputs, which can lead to retrieved information being drowned out. Aside from technical limitations, the most important one is that the overhead makes it an expensive approach for

consumers. Neither setting up a local retrieval system nor doing this in the cloud is trivial from hardware and economical perspectives. Similarly, while fine-tuning on private codebases has been successful for many large corporations [15], is it not practical to do this on a user’s own hardware, nor is it practical to do this on the cloud for every user. Fine-tuning suffers from this even more so than RAG, as updating a knowledge base is far more efficient than retraining as the underlying codebase changes.

Previous work by Tu, Su, and Devanbu has shown that using a global n-gram model for general programming abilities alongside a local n-gram model for project-sensitive code can drastically improve code completion performance in new environments [1]. As small n-gram models can be trained on individual codebases in a matter of seconds, it is possible to apply such approaches to on a user’s own hardware to adapt code completion to their own code. Nevertheless, such a global-local hybrid has not been explored when using a global LLM with a local n-gram model. This thesis aims to find an efficient and effective way to incorporate project context into code completion models at minimal cost. Moreover, it focuses on replicating and building upon the results of Tu, Su, and Devanbu in the context of modern transformer-based LLMs for code. Our experiments and analysis show that modern LLMs are more capable at capturing localness from in-file context alone than local n-gram models are when trained on full codebases. Additionally, we show that targeting specific elements of code that are most likely dependent on the project context can lead to improved performance. Furthermore, we demonstrate that masking techniques combined with Bayesian inference lead to the best overall performance. The performance improvement of our hybrid model is limited, showing that hybrid models incorporating n-grams may not be the best approach for adaptive code completion. Nevertheless, the low computational overhead of our approach makes it possible to incorporate in existing code completion solutions to improve code completion performance by a slight margin.

Our contributions are as follows:

- We rigorously investigate global-local hybrids based on Bayesian inference using dynamic weighting and interpolation techniques, and demonstrate that modern LLMs are superior to local n-gram models in capturing localness;
- We show that local n-gram models focused on project-sensitive parts of code such as identifiers (TIC) can lead to limited performance improvements when combined with a global LLM;
- We show that combining masking techniques can improve accuracy, and that retriever-ranker systems form a good basis for hybrid models;
- We expose shortcomings in LLMs’ awareness of their limitations;
- We publish our source code and datasets, including dataset creation, evaluation, benchmarking, and adaptable adaptive code completion architecture [16].

This thesis is structured as follows: Chapter 2 outlines relevant background information, including information model architectures and generation techniques relevant to this study; Chapter 3 highlights the research objectives and proposed approach to reaching these objectives; Chapter 4 introduces the experimental setup and details the dataset and code architecture used for our experiments; Chapter 5 replicates the approach by Tu, Su, and Devanbu using global LLMs, showcasing differences in the abilities of n-gram models and LLMs; Chapter 6 details improvements upon the previous approach by targeting identifiers; Chapter 7 investigates the overhead of our approach, showing that we can improve code completion without substantially increasing latency; Chapter 8 discusses the implications and limitations of our findings; Chapter 9 concludes the thesis; and lastly, Chapter 10 outlines potential avenues for future research.

# 2

## Background and Related Work

This chapter outlines a high-level overview of the context in which this thesis was written and provides a comprehensive overview of related works.

Section 2.1, Section 2.2, and Section 2.3 provide an introductory overview to serve as foundational knowledge to interested readers. Section 2.4 contains a comprehensive overview of related works.

### 2.1. Large Language Models

Large Language Models (LLMs) are a machine learning models trained on big corpora of text to facilitate tasks that require natural language understanding and generation capabilities. These models can be trained to perform a wide range of tasks, including machine translation, sentiment analysis, machine summarization, question answering, and text generation. Many such tasks have analogues in software engineering, such as code generation, code completion, and code summarization. While there are many different types of LLMs, such models generally involve tokenization, followed by a series of neural network layers that process the tokens. The output of a model can be trained to perform a specific task, e.g., autoregressive generation models are trained to classify the next token, while sentiment analysis models are trained to classify the sentiment of a text. Autoregressive models are typically applied recursively, where the output of the model is appended to the input to generate long token sequences.

#### 2.1.1. Tokenization

Tokenization converts a sequence of characters into a sequence of tokens. This process is essentially a form of compression that reduces the size of the input data, allowing LLMs to process larger inputs.

##### Byte-Pair Encoding

A common tokenization strategy is byte-pair encoding [17]. Byte-pair encoding starts with an initial vocabulary, and introduces new tokens consisting of a pair of existing tokens based on their frequency in the training data. This process is repeated until the vocabulary reaches a fixed limit. While vocabulary sizes differ between tokenizers, the vocabulary size tends to be in the tens of thousands [3, 4, 18, 19].

The initial vocabulary of a byte-pair-encoding-based tokenizer is typically set to all possible byte sequences. This allows the tokenizer to map arbitrary byte sequences to tokens, preventing the possibility of untokenizable sequences.

##### Token Embeddings

A key component of language understanding is the ability to map tokens to a high-dimensional vector representation. There are many facets that define the meaning of a word or token. Conveying this meaning to a machine is challenging but can be learned by training models on large text datasets [20]. To facilitate token understanding, tokens are mapped to a vector embedding space where each dimension represents some aspect of the token's meaning. The exact meaning of each dimension is not relevant as long as models can learn how to interpret the different dimensions of the vector. It is important to use a high enough dimensionality to fully capture the meaning of a token. A typical choice in

modern models is to use 768-dimensional vectors [3, 4, 18, 21, 19]. Token embeddings are a crucial part of modern language models and form a central component of language understanding.

### 2.1.2. The Transformer Architecture

The transformer architecture, introduced by Vaswani et al., is a modern architecture for sequence processing that uses an attention mechanism to efficiently and effectively capture interdependencies in sequences. Previous models, such as Recurrent Neural Networks [22] (RNNs) with Long Short-Term Memory [23] (LSTM) or Gated Recurrent Units [24] (GRUs) face difficulties in capturing long-range dependencies. A large text may, for instance, refer to the opening section of the text in its closing sentence. To properly understand the meaning of this reference, the model must be able to capture the semantic meaning of the section's content, rather than just the section name. Similarly, words within the same sentence are interdependent and change in meaning depending on each other. E.g., words like 'bank' can refer to a financial institution or land running along a river. The true meaning of the word only becomes clear from the surrounding context.

#### The Attention Mechanism

While LSTMs and GRUs process sequences of tokens one at a time, the transformer architecture's attention mechanism processes all tokens in parallel. This not only makes the model more efficient during training and inference, but also improves capabilities at understanding long-range dependencies. While LSTMs and GRUs naturally focus on more recent tokens, the transformer architecture is not naturally aware of the position of tokens in its input sequences. To address this issue, tokens are passed through a positional encoding layer that adds this extra information to token embeddings. Overall, the attention mechanism is a key element of the transformer architecture that has caused a shift in the field of natural language processing.

#### Encoders and Decoders

Transformer models typically consist of an encoder or a decoder, or both. Encoders are typically used for understanding tasks that require a complete sequence of tokens as input. A typical example is sentiment analysis, where the encoder is combined with a classification layer to label the input. Decoders, on the other hand, are typically used for autoregressive generation tasks, as decoders are only able to read the input up to a certain point. ChatGPT, for instance, is a decoder-only model; all required information is located in the input, making it suitable to use a decoder-only model. A combination of encoders and decoders is also possible for tasks that require both understanding and generation. For instance, machine translation requires understanding the input, and generating the output.

In the context of software development, decoder-only models are commonly used for code completion, whereas encoder-decoder models are used for code generation and documentation generation. Encoder-only models can be used for tasks like bug localization.

### 2.1.3. N-gram Models

N-gram models are statistical models that use *n-grams* to probabilistically predict the next token [25]. The simplest way to estimate conditional probabilities of possible next tokens is using the formula displayed in Equation 2.1. However, there are different methods to estimate  $p(t_i|t_{i-n:i-1})$  that incorporate *smoothing* or *interpolation*. N-gram models are constrained by the n-gram order  $n$ , as increasing values of  $n$  leads to exponentially larger models. This makes it difficult for n-gram models to accurately predict long sequences, and makes it impossible for them to capture long-range dependencies in text.

$$p(t_i|t_{i-n:i-1}) = \frac{p(t_i|t_{i-n:i-1})}{\sum_{t_{i-n:i-1}} p(t_i|t_{i-n:i-1})} \quad (2.1)$$

N-gram models frequently run into *out of vocabulary* (OOV) issues. OOV issues arise when the model is unable to predict the next token because it has never seen the input before. This issue is especially common when using basic n-gram models that look up the input verbatim. There are several ways to address this issue, which typically involves adjusting the probability distribution. The simplest approach is to use a *back-off* strategy, where the n-gram model will fall back to a lower order  $n - 1$  if it is unable

to find the input. Alternatively, *smoothing* and *interpolation* methods can be used to assign probability mass to unseen tokens.

While these methods can alleviate some OOV issues, n-gram models are still far less capable than modern large language models.

## 2.2. White-box Generation Techniques

White-box generation techniques are generation techniques that are aware of, or require cooperation with the underlying language model. Examples of such techniques include *chain-of-thought* prompting, as the underlying language model must be able to be able to ‘reason’ about its own output; models trained on solely code data will typically not be able to do this, while models trained to be able to follow instructions are able to do so [26, 27]. White-box generation techniques may be more effective than black-box generation techniques as the underlying language model must be aware of (part of) the generation objective.

### 2.2.1. Retrieval-Augmented Generation

A common white-box generation technique is *retrieval-augmented generation*. RAG provides additional context to the LLM by retrieving documents relevant to the prompt [11, 12, 13]. Additional information can either be retrieved dynamically by the underlying language model (e.g., through OpenAI’s *tools*) or be added automatically through information retrieval (IR) based approaches [11, 28]. In the context of code completion, this could include code snippets, documentation, or even full files. While it is possible for RAG to be used as a black-box pre-processing technique, in its conception a query encoder, retriever, and LLM were trained in tandem [11]. I.e., the language model was trained on outputs of the retriever, making this a white-box approach. Nevertheless, retrieval-augmented generation reduces the need for retraining as the information available to the retriever can be swapped out as needed. In the context of code, RAG is especially useful for code completion in private projects: such codebases will not have been available for training, leading to frequent hallucinations.

### 2.2.2. Downsides of White-Box Enhancements

While white-box generation techniques can be very effective, their reach is limited by the availability of compatible language models. Such constraints do not restrict black-box generation techniques by design. Such techniques make very few assumptions about the underlying LLM to support a broad range of models without the need for retraining or fine-tuning. While this makes them ideal for widespread use, the lack of cooperation with the underlying model may lead to worse performance compared to white-box techniques.

## 2.3. Black-Box Generation Techniques

Black-box techniques generally present themselves as either pre-processing or post-processing approaches, or both. In the context of code completion, for instance, a simple black-box technique involves prepending code snippets to the prompt as a pre-processing approach [29, 30, 31, 32, 33]. Alternatively, it is possible to constrain the output of language models to a specific format as a post-processing technique using tools like Outlines [34]. The latter can, for instance, be used to ensure that the output is valid Python code, JSON, or some other structured format like certain classification labels. This approach has shown use in combination with static analysis tools to ensure that only valid class methods can be used and to ensure that predicted identifiers used as argument match the parameter type [35].

Lastly, Tu, Su, and Devanbu explore a post-processing approach that combines a global n-gram model with a local n-gram model. The global model is trained on a large corpus of code, facilitating general coding capabilities, whereas the local model is trained on the local project, allowing it to help with code that requires knowledge of the project [1].

## 2.4. Related Work

Several previous works have investigated methods for adapting LLM-based code completion models to new codebases. Most of these techniques rely on relatively expensive processes, such as training



a retriever.

### 2.4.1. Retrieval-Augmented Generation

A common approach to adapting LLMs for code completion to new projects is RAG [12, 13, 11]. For instance, Lu et al. introduce ReACC, a framework that enhances code completion quality through semantic search. The authors use a dual-encoder setup to find related code fragments, which are then simply prepended onto the model input. Results show significant improvements in code completion quality. In certain cases, a fine-tuned CodeGPT [36] using RAG outperforms GitHub Copilot, which is based on the much more powerful Codex [5]. Similarly, Guu et al. use a RAG approach for the question answering task. Rather than using semantic search, this work proposes training a retriever that retrieves documents that will improve the perplexity of the LLM. They show that this approach can outperform previous techniques and improve the modularity of models by separating knowledge from language generation. REDCODER [38], a framework similar to ReACC, once more shows that RAG can be used to improve code generation performance. DocCoder[33] takes a different approach by retrieving documentation rather than code snippets. Once more, this RAG approach is shown to be successful for Python and Bash code. In fact, the results indicate that documentation is better at providing additional context than code snippets when using the REDCODER architecture.

### 2.4.2. Adaptive Code Completion with White-Box Retrieval

While previous approaches use retrievers as a black box, there are alternative approaches that use language-specific features to intelligently represent project context to make it possible to collect relevant code snippets. Additionally, there are also alternative approaches to RAG that adapt model outputs rather than model inputs. Khandelwal et al. use  $k$ -NN on embeddings to find semantically similar tokens and use Bayesian inference to combine this with code completion model outputs [39]. This approach allows for seamless integration with existing code completion LMs as no fine-tuning on inputs that include retrieved snippets is required. Results show improved performance compared to all baselines. Similarly, Tang et al. propose using a  $k$ -NM ( $k$ -nearest mistakes) LM to integrate domain knowledge with language modeling [40]. Rather than using  $k$ -NN to find suitable tokens to similar inputs,  $k$ -NM aims to find tokens where LMs frequently make mistakes. This can be used to prevent making similar mistakes during inference. Similar to Khandelwal et al.,  $k$ -NM outputs are combined with model outputs using Bayesian inference, leading to seamless integration with existing code completion LMs. Results show that this approach can significantly improve performance and is more effective than Bayesian inference with  $k$ -NN or ReACC retrievers. Ding et al. introduce CoCoMIC, a framework that builds a project context graph to retrieve relevant context and uses relevant context to enhance code completion models [41]. Retrieved code snippets are represented as embeddings and included in a fine-tuned model. Results show significantly improved performance on Python code. Shrivastava, Larochelle, and Tarlow introduce RPLG, a prompt generator that creates prompts based on a local code context within a repository [31]. RPLG first determines where context should be retrieved from, after which the type of context is determined (e.g., identifier, literal, method name). Results show significant improvements when applied on Codex and different OpenAI models. Agrawal et al. take a different approach and, similarly to our method, adjust the output of the LLM based on the project context [35]. Rather than creating a local model, the authors opt to use static analysis to enforce the correctness of the code. This method can ensure that the model does not hallucinate class methods and uses variables with types that match parameter types. The results show that this approach increases accuracy significantly, increases the compilation rate of outputs, and allows smaller code LLMs to perform at the level of larger code LLMs. Similarly, STALL+ [42] integrates static analysis into LLM-based code completion at the during decoding and as post-processing technique, leading to significant improvements in line-level and identifier-level accuracy. The results also show that using static analysis works best for statically typed languages.

## Research Objectives and Approach

Previous work by Tu, Su, and Devanbu showed that combining large n-gram code models with smaller n-gram models trained on local code can lead to improved code completion performance in unknown environments [1]. While this approach was effective when combining two n-gram models through Bayesian inference (BI), this approach has not been explored when combining a global LLM with a local n-gram model. Nevertheless, hallucinations remain a common problem when using LLMs for code completion in new contexts like private codebases, showing the need for better adaptation capabilities. Local n-gram models can be trained in realtime, making them ideal for on-device use in real-world scenarios. As such, we investigate **RQ1: To what extent can n-gram models trained on local code support LLM code completion in new environments through Bayesian inference?**

LLMs are excellent at writing syntactically correct code but typically struggle at writing code that fits within context contained in external files. Not all code requires knowledge of project context; keywords to create new classes or functions are easy to predict, and isolated logic that does not interface with other parts of the codebase can be generated by an LLM from solely informative function name alone. As such, the LLM does not require the guidance of an n-gram model trained on local context for every prediction. Limiting the frequency at which the n-gram is applied can help reduce performance degradation by letting the LLM focus on writing correct code while the n-gram model only helps when knowledge of project context is needed. We investigate which parts of code depend on project context through **RQ2: Which parts of code benefit most from knowledge of project context?**

Our findings indicate that identifiers are sensitive to project context, making it possible to use a local n-gram model trained on identifiers to guide the global LLM towards syntactically and semantically correct code. To build on top of the work by Tu, Su, and Devanbu we explore different approaches to further improve code completion performance by focusing on identifiers. We introduce Targeted Identifier Completion (TIC), and investigate **RQ3: To what extent can TIC improve code completion?**

We consider three different approaches to improving TIC:

- **RQ3.1:** To what extent can masking techniques improve TIC accuracy?
- **RQ3.2:** Which components of masking approaches are most dependent on project context?
- **RQ3.3:** To what extent can masking combined with Bayesian inference improve TIC accuracy?

To reduce the overhead of the hybrid model, we investigate the possibility of conditionally applying TIC to prevent the use of the n-gram model when the LLM is capable of predicting an identifier on its own. Built-in standard libraries, for instance, do not rely on knowledge of the local codebase and do not benefit from the help of a local n-gram model. Incorrectly preventing activation of the n-gram model can compromise any accuracy gains that the n-gram model may provide, showing the importance of a balanced approach. We examine **RQ4: To what extent can conditional application of TIC reduce computational overhead without compromising accuracy?**

Furthermore, we determine whether TIC can be applied in real-world settings by benchmarking the

overhead of TIC compared to the baseline, answering **RQ5: To what extent does TIC increase the latency of code completion?**

We answer these research questions using a hybrid model consisting of a global LLM and a local n-gram model. While we use different approaches throughout this thesis, the LLM is always responsible for general coding skills due, while the local n-gram model is responsible for specific details in the context of the project in which the hybrid model is used.

# 4

## Experimental Setup

Black-box generation techniques are widely applicable methods that have the potential to improve code completion performance at scale. While white-box techniques can be more powerful, black-box techniques do not rely on underlying models being aware of the technique, leading to practical improvements that remain relevant as new LLMs are released.

This chapter outlines the dataset and LLM used to construct our hybrid model consisting of a global LLM and local n-gram model. Moreover, it details the setup used to perform, evaluate, and analyze our experiments.

### 4.1. Dataset

We use the *Long Code Arena* benchmark [43] to assess the performance of our hybrid model. Long Code Arena is a benchmark suite created to assess the performance of models that use long context windows. Previous works primarily focus on local context within the current file, but increases in input length have made it possible to include project-level context within prompts of newer models. The Long Code Arena contains benchmarks for library-based code completion, CI build repair, project-level code completion, commit message generation, bug localization, and module summarization. We use the small variant of the project-level code completion benchmark, which consists of 144 Python test files with metadata indicating in which lines code completion can benefit from in-project data. This benchmark was constructed by inspecting the Git history of repositories and using parent commits as context for code added in new commits.

We apply processing steps to the benchmark to use files from the same commit as context, rather than using cross-commit context. We create three variants of this benchmark for our different experiments to support three different code completion tasks; the first invokes code completion after the first non-whitespace token of the line, the second invokes code completion on identifiers, and the third invokes code completion on function invocations. We only consider lines where in-project context is relevant to be able to measure the performance of our hybrid model incorporating project context. The different dataset variants are available through our replication package [16].

### 4.2. Model

We use Salesforce’s CodeGen-350M-mono [8] as our model of choice for our experiments.<sup>1</sup> This model is a small variant of the CodeGen model based on the CodeGen-350M-multi model, but trained on 71.7B additional tokens of Python code from the BigPython dataset [8]. The chosen model is ideal as it is small and cheap to run whilst also performing relatively well against our test set. Additionally, since the chosen Long Code Arena benchmark excludes commits before January 1st, 2022, there is no overlap between the test samples in the benchmark and the model. We apply this model using greedy decoding throughout our experiments.

---

<sup>1</sup><https://huggingface.co/Salesforce/codegen-350M-mono>

## 4.3. Metrics

We use two metrics to evaluate the performance of baselines and our hybrid model variants:

- **Accuracy.** We use accuracy to measure the proportion of samples that the model correctly predicts in its entirety. In cases where a test sample requires multiple tokens to be prediction, all predictions must be correct for the sample to be 100. If a single token is incorrectly predicted, the entire prediction is incorrect, leading to an accuracy of 0. Throughout our experiments we average the accuracy over all test samples in the datasets.
- **Next-token accuracy.** We use next-token accuracy to provide a more nuanced measure of a model's performance. It is theoretically possible for accuracy to be 0 despite being able to correctly predict 90% of all tokens. For this reason we include next-token accuracy. Next-token accuracy works similarly to accuracy, but instead of considering accuracy over the entirety of a sample's ground truth, it only considers the next token. To calculate this measure, we fix the input of a sample before predicting the next token. I.e., if there are more tokens to be predicted, we expand the left context by moving a token from the ground truth to the input. This way we compute the accuracy over all tokens in the ground truth, leading to a more nuanced measure of a model's performance compared to accuracy.

While many previous studies opt to use NLP-based metrics such as BLEU [44] or ROUGE [45] to evaluate code completion models, these metrics have been shown to poorly measure model performance, often resulting in statistical insignificance despite gaps in metric values [46]. Observed improvements in raw accuracy are less prone to statistical insignificance due to the exact nature of this metric.

Throughout our experiments, our primary goal is to improve the *accuracy* of our model. However, we use *next-token accuracy* to compare and choose different hyperparameter values and use its results to inform our model design, as it is a more granular measure.

## 4.4. Architecture

We design our hybrid global-local system with modularity and extensibility in mind. This chapter highlights the architecture of our code and how it can be extended to support different interpolation techniques, n-gram models, and more.

We leverage the fact that Python supports multiple inheritance to use a mixin-like approach to modularity. We define a base class `AdaptiveCodeCompletion` that provides abstract method signatures for different features of our system, along with a concrete `setup` method and a concrete `get_token_probs` method that uses all abstract methods to compute the next-token probability in an adaptive manner. The abstract methods present on the `AdaptiveCodeCompletion` class are as follows:

- `n_gram_token_limit()`: Returns the maximum number of tokens (or None) that can be used in the n-gram model.
- `select_n_gram_files(dataset_file_system, source_file_path)`: Select files to use for training an n-gram to be applied to the given source file.
- `extract_n_gram_token_sequences(text)`: Extracts the n-gram token sequences from the given text.
- `should_activate(input_tokens, next_token_probs)`: Determines whether the n-gram model should be activated for the given input tokens and next-token probabilities.
- `get_n_gram_token_input(input_tokens)`: Extracts the n-gram token input from the given input tokens.
- `get_n_gram_distribution(input_tokens)`: Returns the n-gram distribution for the given input tokens.
- `mix_probability_distributions(n_gram_dist, llm_probs, n_gram_input)`: Mixes the n-gram distribution with the LLM probabilities for the given n-gram input.

Different parts of our system implement different subsets of these abstract methods. The final concrete class composes multiple partial concrete classes to construct a class that implements all abstract methods. This approach makes it easy to implement features in isolation and to combine them in different

ways. For instance, changing from fixed-weight to dynamic-weight Bayesian inference is as simple as swapping a class that implements the `mix_probability_distributions` method. Similarly, changing the granularity of the n-gram model (i.e., whether it uses identifiers, function invocations, or all tokens) is simply by swapping to a different class implementing the `get_n_gram_token_input` (used during inference) and `extract_n_gram_token_sequences` (used during training) methods.

Our composable approach also makes it trivial to connect our system to Hugging Face's `transformers` library. To facilitate this, we implement a `StrategyLogitsProcessor` class that implements the `LogitsProcessor` interface from `transformers`. This class receives a concrete `AdaptiveCodeCompletion` instance, a tokenizer, and a list of input and output tokens, and applies the adaptive code completion strategy to every input in a batch. Note that the list of output tokens is only necessary for the collection of performance metrics and is not required for real-world deployment. Supplying this `StrategyLogitsProcessor` the `generate` method of a Hugging Face `transformers` model instantly enabled any Hugging Face model to integrate with our n-gram model, making it simple to integrate our system into any model that is available on the Hugging Face Hub. Additionally, our system is compatible with all other options available in the `generate` method, such as different temperatures and sampling strategies.

We showcase a complete UML class diagram of our system in Appendix A (Figure A.1).

# 5

## Is Localness all You Need?

Before the rise of neural NLP models such as recurrent neural networks and transformers [2], language modeling was a pure statistical modeling task. The probability distribution over possible next tokens was modeled by considering exactly the distribution in the training data: the last  $n - 1$  tokens of some arbitrary input sequence has are used as a prefix to find continuation tokens, after which normalizing yields a probability distribution of next tokens  $p(t_i | t_{i-n:i-1})$ . Such statistical models are referred to as n-gram models. While sparseness is a clear issue, especially for higher values for  $n$ , interpolation and smoothing techniques can help models approximate in such situations. In “On the Localness of Software” by Tu, Su, and Devanbu [1] show that human code exhibits the notion of *localness*: there are certain regularities that can be exploited by smart models to write code that accurately reflects the context of the code being written. Such local features may be common variable names, commonly used internal libraries, or syntactical preferences such as the positioning of braces. Given that n-gram models generalize over large corpora of data it is clear that such local features are generally left unexplored.

Tu, Su, and Devanbu explore a hybrid approach to soften the difference between an average program and a program within an unknown new environment. This approach utilizes a global n-gram model for general coding skill in conjunction with a local n-gram model that captures the local qualities of the codebase. The *cache model* is constructed by combining the two probability distributions using a simple weighted sum (i.e., Bayesian inference). The weight  $\lambda$  is set as a function of the local model; a higher prefix frequency in the local model leads to a lower  $\lambda$ , and a higher bias towards the local model as it is deemed more reliable. Equation 5.1 shows how the weight is determined, where  $\gamma$  is a *concentration parameter* in the range  $[0, \infty)$ , and  $h$  is the prefix frequency in the local model.

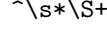
$$\lambda(h) = \frac{\gamma}{\gamma + h} \quad (5.1)$$

Tu, Su, and Devanbu show promising results; the hybrid model improves raw performance by around 14%. Despite these promising results on global-local n-gram hybrids, similar setups using neural code models have not been elaborately explored. It stands to reason that this idea is not scalable when considering neural models, especially when the codebase changes frequently, resulting in the need of frequent re-training. Yet it remains a possibility to combine a global neural code model with a local n-gram model, which we investigate in this thesis.

### 5.1. Exploring LLM + N-gram Bayesian inference

In this section, we investigate Bayesian inference, dynamic weighting (dynamic Bayesian inference, DBI), and smoothing techniques to explore whether the ideas of Tu, Su, and Devanbu transfer to a hybrid LLM-n-gram setup.





**Figure 5.1:** Regular Expression for determining the Cursor Position in Line Code Completion ( $\backslash s$  matches whitespace characters,  $\backslash S$  matches non-whitespace characters)

### 5.1.1. Training

Similar to Tu, Su, and Devanbu, we construct a local  $n$ -gram model by tokenizing project files, and extracting subsequences of length  $n$ . We use the LLM’s tokenizer to tokenize file contents, after which we use a sliding window to extract subsequences of length  $n$ . We set  $n = 3$  for our experiments to strike a balance between supporting verbatim code duplication and supporting general programming. As  $n$ -gram models of this size are quick to train, we perform training live as we iterate over the test set. We train a new  $n$ -gram model for every test sample, making sure to exclude the file in which the model is being invoked from the training data to prevent bias. We incrementally add and remove model inputs to the  $n$ -gram model during inference rounds for the same test sample. This ensures that the  $n$ -gram model is able to use any in-file context located before the cursor. Our incremental approach serves to reduce computational overhead. I.e., at the start of every round of inference, the model input is inserted into the  $n$ -gram model, after which this input is removed again at the end of the inference round. This makes it possible to maintain an up-to-date  $n$ -gram model without needing to re-train it from scratch for every round of inference.

### 5.1.2. Dataset Creation

We create a dataset based on the Long Code Arena Project-Level Code Completion benchmark [43]. The dataset is constructed as a file system that contains both project context and test samples, i.e., locations at which to perform code completion as well as the ground truth to predict. The Long Code Arena benchmark provides line numbers for lines that require knowledge of project context to be completed. We construct test samples by splitting the line after the first non-whitespace using the regular expression displayed in Figure 5.1.

### 5.1.3. Bayesian Inference

The core principle behind the global-local hybrid model is Bayesian inference, i.e., we construct an ensemble model by blending the probability distributions produced by individual models. Equation 5.2 shows what this process entails, with  $w_{ngram}$  being the weight attributed to the probability distribution generated by the  $n$ -gram model. Selecting an appropriate weight is essential to improving code completion performance.

$$p_{hybrid}(t_i | t_{i-n:i-1}) = p_{LLM}(t_i | t_{i-n:i-1})(1 - w_{ngram}) + p_{ngram}(t_i | t_{i-n:i-1})w_{ngram} \quad (5.2)$$

We conduct three experiments; first, we run the hybrid global-LLM and local- $n$ -gram with  $w_{ngram} = 0$ . This experiment serves to establish the baseline performance of the chosen LLM against our dataset. Next, we run an inference-time grid search over the domain  $W_{ngram} = \{0.0, 0.1, \dots, 0.9, 1.0\}$  to find the theoretical upper bound (UB) for accuracy. Note that this search runs for every round of inference, thus simulating an oracle that can dynamically predict the optimal weight. Lastly, we set  $w_{ngram}$  to an appropriate constant based on the results of the previous experiment to establish the performance gains that can be achieved using a static weight.

During these experiments we capture the *next-token accuracy* of the model. This quantity is collected over all ground-truth tokens, i.e., for every token in the ground truth, we fix the left context and prefix of the ground truth and capture the models’ ability to predict the next token. This number informs us over a model’s general ability, making it possible to make actionable decisions based on this number. We use this number rather than the accuracy over the ground truth, as the latter discards the performance over all tokens the moment a single token is predicted incorrectly, and also is not invariant to the length of the ground truth.

**Experiment 5.1: Baseline LLM Code Completion** As shown in Table 5.1, the baseline accuracy is 42.11, meaning that the ground truth was correctly predicted in 42.11% of all test samples. The next-token accuracy is higher at 81.67%, as can be seen in Figure 5.2 for  $w_{ngram} = 0.0$ , meaning that local

Experiment	Accuracy
Baseline	42.11
UpperBound	44.21
$w = 0.4$	41.05

Table 5.1: BI Hybrid Model Accuracy

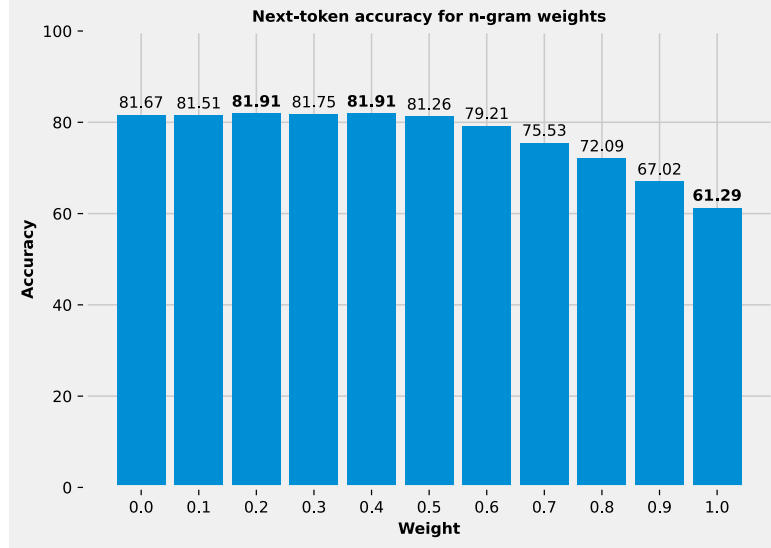


Figure 5.2: Next-token Accuracy for BI Hybrid Model

n-gram model was not used. The difference between these two numbers is explained by the fact that the ground truth typically consists of more than one token, leading to a lower accuracy over the full ground truth.

**Experiment 5.2: UB on BI Accuracy** The theoretical upper bound on accuracy comes in higher than the baseline at 44.21% (Table 5.1). This indicates that Bayesian inference may be able to help improve code completion performance, but by a relatively small margin. The average next-token accuracy over different values for  $w_{ngram}$  is displayed in Figure 5.2. For most weights the next-token accuracy does not improve far beyond the next-token accuracy of the baseline, indicating that performance gains are likely limited in this scenario. Additionally, we observe that higher weights lead to considerable performance degradation. This indicates that the n-gram model on its own is not nearly as proficient at code completion as the LLM and highlights that selecting an appropriate weight is of utmost importance.

**Experiment 5.3: BI Accuracy with  $w_{ngram} = 0.4$**  Based on Figure 5.2 we set  $w_{ngram} = 0.4$  for this experiment. This configuration leads to an overall accuracy of 41.05%, which is below the baseline performance (Table 5.1). Clearly, n-gram Bayesian inference using a static weight is not an appropriate way of improving the performance of an LLM when used in new contexts. We theorize that the fact that the local n-gram model competes with a global LLM makes improving performance especially difficult; the LLM is so proficient that it may fare better alone than with the help of a local model that *does* capture local information, but performs poorly overall. This issue does not present itself when using an n-gram model for both the global and local model, as they are more evenly matched.

#### 5.1.4. Dynamic weighting

We continue our experiments by incorporating dynamic weighting using Equation 5.1. We perform two more experiments; one to establish a theoretical upper bound and to determine an appropriate constant  $\gamma$ , and one where we use a static  $\gamma$ .

Experiment	Accuracy
Baseline	42.11
UpperBound	44.21
$\gamma = 71$	42.11

Table 5.2: DBI Hybrid Model Accuracy

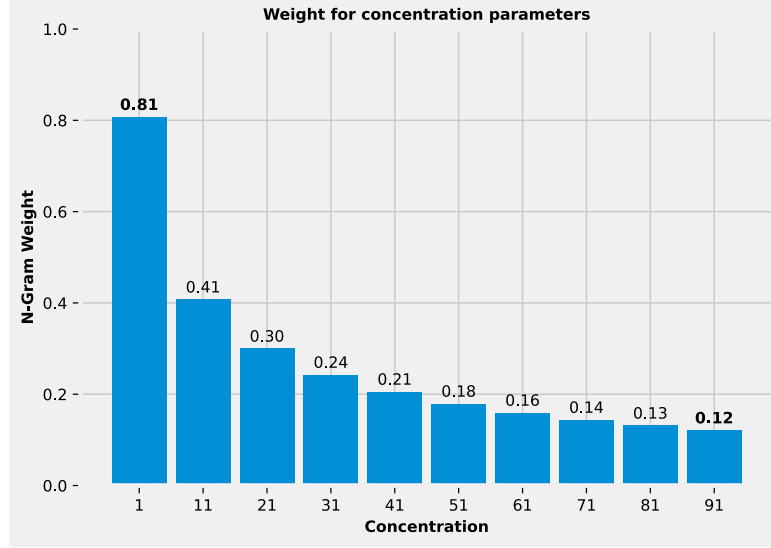


Figure 5.3: DBI Parameter Values Weights

**Experiment 5.4: UB on DBI Accuracy** As displayed in Table 5.2, the theoretical upper bound for dynamic weighted Bayesian inference is an accuracy of 44.21%. This is identical to the upper bound of weighted Bayesian inference. To compare dynamic Bayesian inference to weighted Bayesian inference, Figure 5.3 displays the average weight for different values of  $\gamma$ . As per Equation 5.1, this weight depends on  $h$ , which is the number of times the prefix has been seen in the training data (i.e., project files). This figure looks as expected; as  $\gamma$  increases, the weight becomes more biased towards the LLM. Next, Figure 5.4 displays the average next-token accuracy for different values of  $\gamma$ . This figure clearly indicates that a low  $\gamma$  (i.e., a high  $w_{ngram}$ ) leads to a lower next-token accuracy. The best next-token accuracy is achieved when  $\gamma$  is at its highest, corresponding to the lowest values for  $w_{ngram}$ . This clearly indicates that the local-n-gram model is unsuccessful at helping the global LLM at code completion, even though the n-gram model captures local information that is not captured by the LLM.

**Experiment 5.5: DBI Accuracy with  $\gamma = 71$**  To verify the previous findings, we run our hybrid model with  $\gamma = 71$ . This results in an accuracy of 42.11% (Table 5.2). The fact that the accuracy does not improve over the baseline accuracy confirms that the n-gram model is unable to correct the LLM in context-sensitive code locations.

### 5.1.5. Interpolation and Smoothing

Next, we incorporate interpolation and smoothing techniques to improve the local n-gram model. The approach n-gram models take to model language directly leads to OOV issues: the models are only aware of token sequences that occur in the training data verbatim, and are unable to interpolate between close matches in cases where no exact match is found. This issue is especially prevalent when working with n-gram models on a small dataset, such as our local n-gram models. Methods that aim to alleviate such issues typically use back-off strategies, discounting, smoothing, or interpolate based on assumptions about the distribution of the underlying data. Tu, Su, and Devanbu use Katz’s back-off model [47], but there are many more techniques that have shown mainstream use. We experiment with Katz’s Back-Off, Kneser-Ney, Stupid Back-Off, Witten-Bell, and a simple back-off strategy.

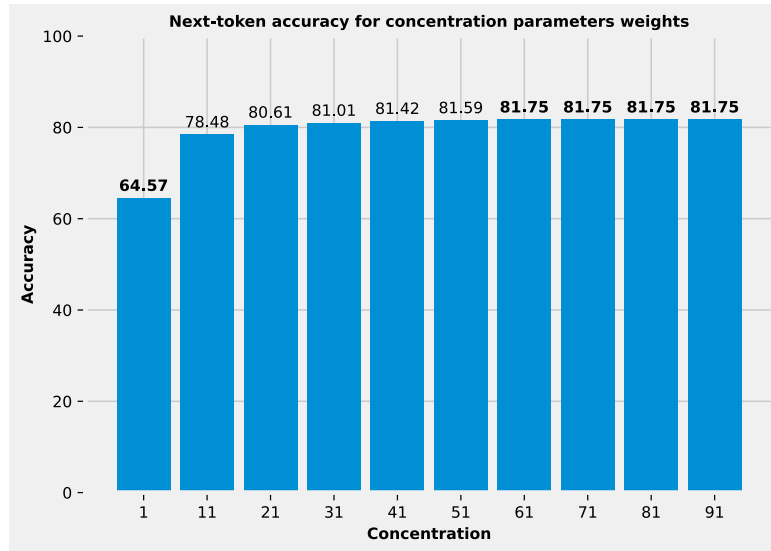


Figure 5.4: DBI Hybrid Model Next-token Accuracy

**Katz's Back-Off** Katz's Back-Off combines relative discounting with backing off to create a dense distribution in cases where the n-gram model is unable to find some prefix. The discount factor  $d$  is typically found through Good-Turing Estimation [48, 49].

**Good-Turing Estimation** Good-Turing Estimation is a method that estimates the probability of unseen tokens with the underlying assumption that the probability of unseen tokens is proportional to the probability of tokens that have been seen once [48, 49]. Additionally, the probability of tokens seen  $r$  times is adjusted proportionally to the number of tokens that were seen  $r$  and  $r + 1$  times ( $N_r$  and  $N_{r+1}$ ). As frequencies  $r$  can often have  $N_r = 0$ , especially for higher values of  $r$ , the Good-Turing Estimation process usually involves creating a regression line for  $N_r$  to be able to provide an estimate for arbitrary values for  $r$ . This regression line is constructed by distributing frequency counts over neighboring frequencies with zero counts, and then fitting a line to the log-log plot of the adjusted counts. This leads to a valid regression line if the slope smaller than  $-1$ .

**Kneser-Ney** Kneser-Ney smoothing uses both absolute and relative discounting to redistribute some probability mass to n-grams with zero occurrences in the training data [50, 51, 52]. A discount parameter is used as both the absolute and relative discount, The discount parameter is typically set to 0.75.

**Stupid Back-Off** Stupid Back-Off is a simple strategy that combines distributions of different order n-grams to create a dense distribution [53]. For every possible next token  $t_i$ , if prefix  $t_{i-n:i}$  is not found in the n-gram model, the distribution  $d^k p(t_i | t_{i-k:i-1})$  for the highest  $k$  (with  $k < n$ ) for which prefix  $t_{i-k:i-1}$  exists in the n-gram model is used instead. The constant  $d$  is a discount factor applied for every token that is stripped from the prefix.  $d$  is typically set to 0.4.

**Witten-Bell** Witten-Bell is a Back-Off based technique that recursively combines probability distributions for n-grams with those for  $(n - 1)$ -grams proportionally to the probability that the prefix is not found but the tail of the prefix is found [51, 54, 52]. It is a simple technique that is straightforward to use as there are no configurable parameters.

**Simple Back-Off Strategy** This strategy finds the longest prefix that is known by the n-gram model. If some prefix is unknown, the procedure is applied recursively on the tail of the prefix. As a base case, the empty prefix maps to a distribution of prior probabilities. This technique is similar to Stupid Back-Off, but rather than backing off on individual next-token candidates  $t_i$ , it backs up based on the prefix alone.

Experiment	Accuracy
Baseline	42.11
UB-None	44.21
UB-Katz	42.11
UB-Kneser-Ney	44.21
UB-Stupid	45.26
UB-Witten-Bell	43.12
UB-Simple	44.21
$w = 0.4$	
None	41.05
Katz	35.79
Kneser-Ney	41.05
Stupid	41.05
Witten-Bell	41.05
Simple	41.05
$\gamma = 71$	
None	42.11
Katz	41.05
Kneser-Ney	42.11
Stupid	42.11
Witten-Bell	42.11
Simple	42.11

**Table 5.3:** BI and DBI Hybrid Model with Interpolation and Smoothing Accuracy

This leads to an output distribution that is sparser than that of Stupid Back-Off. Unlike Stupid Back-Off, this strategy is non-parametric.

We perform two experiments; one to determine the theoretical upper bound when using the interpolation techniques, and one with fixed  $w_{ngram}$  and  $\gamma$  parameters, similar to the previous experiments. To this end, we use the selected interpolation techniques in conjunction with both Bayesian Inference and Dynamic Bayesian Inference. Throughout these experiments we use the defacto standard smoothing/interpolation parameters if applicable, as noted in the technique descriptions. We make a small change to the selected interpolation methods; in the case that any probability mass is redistributed to any tokens with  $p(t_i | \dots) = 0$ , we instead redistribute it proportionally over all known tokens. As our local n-gram models use a limited subset of the large LLM tokenizer’s vocabulary, distributing probability mass to zero-probability tokens would not have a noticeable effect.

**Experiment 5.6: UB on BI Accuracy with Interpolation and Smoothing** As illustrated in Table 5.3, it is possible for smoothing to raise the theoretical upper bound on accuracy beyond the theoretical optimum of 44.21% without smoothing. However, we only observe this improvement for Stupid Back-Off. Kneser-Ney and the Simple Back-Off Strategy do not improve the upper bound but also do not cause any regression. In contrast, Katz’s Back-Off and Witten-Bell reduce the theoretical maximum accuracy slightly, but do not cause regression below the baseline performance. The notable performance deterioration when using Katz’s Back-Off strategy can likely be attributed to the fact that we were unable to use smoothed counts as per the strategy’s specification. Katz’s Back-Off model uses Good-Turing Estimation to determine the discount factor. Good-Turing Estimation in turn uses linear regression on a log-log plot of frequencies and the frequencies of frequencies for interpolation, however the regression line is only valid when its slope is no larger than -1. In our experiments we observed that our data consistently leads to invalid regression lines, leading us to omit smoothing. While this is generally acceptable for low frequencies, the sparsity of the frequencies of low-frequencies leads to unrepresentative values for high frequencies. The fact that smoothing was not applicable to our n-gram model indicates differences between the n-gram token distribution and a typical distribution expected

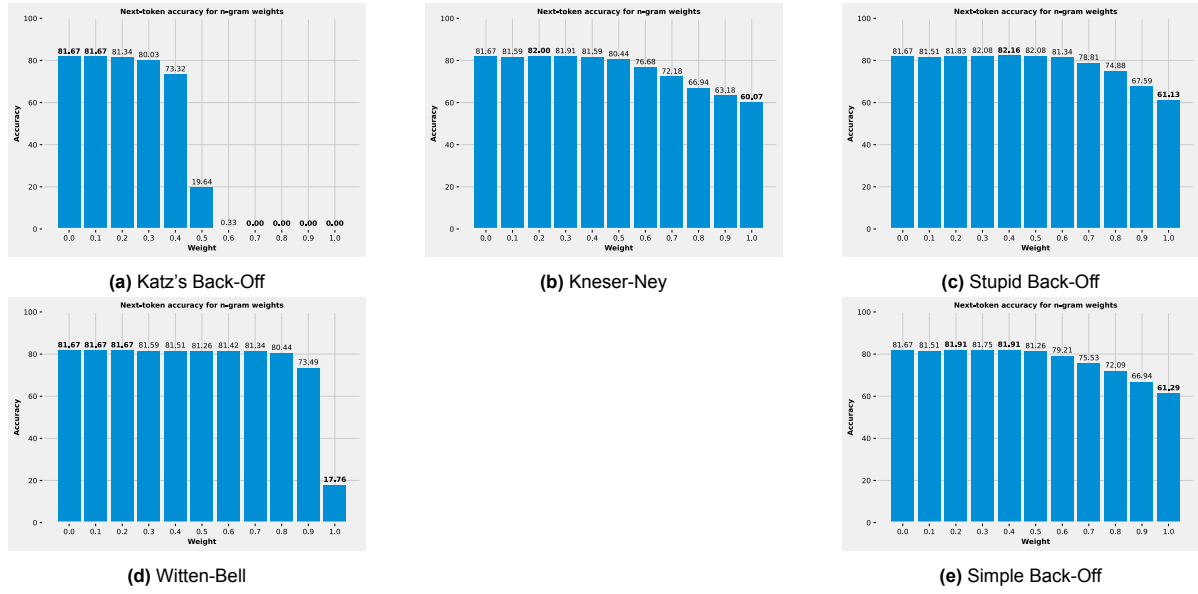


Figure 5.5: BI Hybrid Model Next-Token Accuracy (Interpolation and Smoothing)

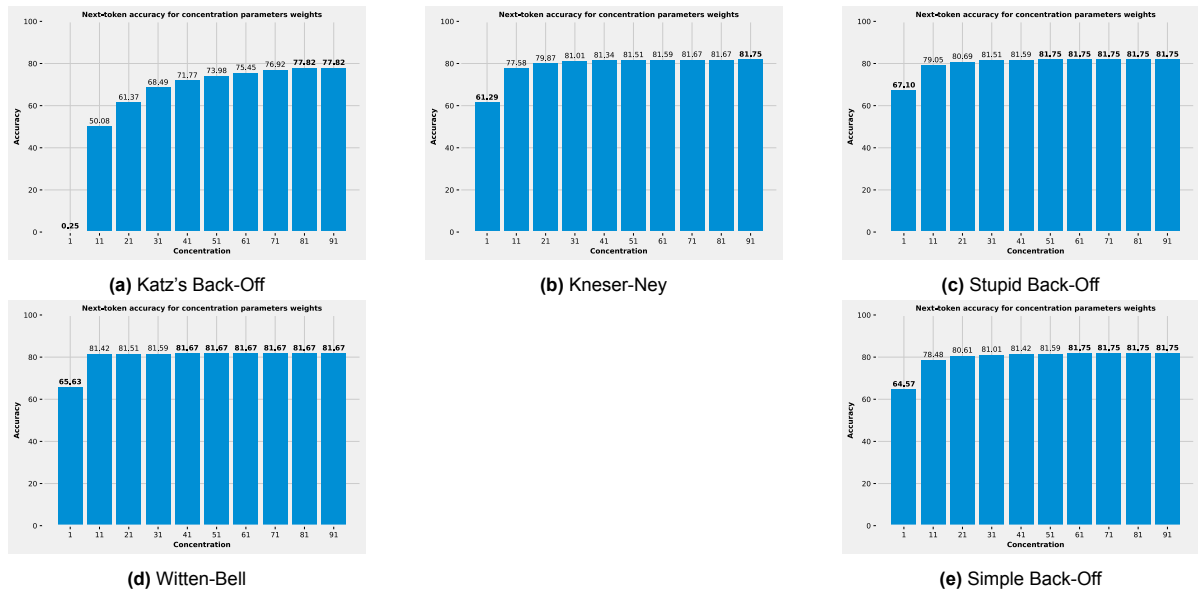


Figure 5.6: DBI Hybrid Model Next-Token Accuracy (Interpolation and Smoothing)

by Good-Turing estimation. This could be explained by differences in tokenization, vocabulary size, training data, or by the fact that our n-gram model is relatively small.

We observe different next-token accuracies for the different smoothing methods as depicted in Figure 5.5 and Figure 5.6. First, it is clear that Katz’s Back-Off is not effective: an n-gram model using Katz’s Back-Off yields to a next-token accuracy of 0%. Of the remaining strategies only Witten-Bell was unable to improve the next-token accuracy for any  $w_{ngram}$ . Simple Back-Off, Kneser-Ney, and Stupid Back-Off are able to improve the performance of our hybrid model at a next-token granularity to different extents, with Stupid Back-Off being able to provide the biggest boost in performance. Nevertheless, the next-token accuracy improvement is extremely small compared to the baseline (+0.49%pt.). We observe that fixed-weight Bayesian inference appears to be superior to dynamic Bayesian inference when combined with smoothing methods in this context. No smoothing method was able to improve performance when combined with dynamic Bayesian inference. It is clear that dynamic weighting (Equation 5.1) is the reason for this observation, as the smoothing is identical between the two experiments.

Overall, we observe one isolated case where interpolation lifts the upper bound performance beyond the upper bound performance without interpolation. Additionally, we find consistent results between line completion accuracy and next-token accuracy, indicating that Stupid Back-Off is most effective, followed by Simple Back-Off and Kneser-Ney. Nevertheless, interpolation nor smoothing increases the upper bound performance, reinforcing that even with smoothing or interpolation the n-gram model is not successful at improving the code completion abilities of the global LLM.

**Experiment 5.7: BI Accuracy with Interpolation and Smoothing using  $w_{ngram} = 0.4$**  To further verify our findings, we repeat our experiment on different interpolation techniques using a static  $w_{ngram} = 0.4$ . As indicated by Table 5.3 this does not improve performance compared to the baseline. In all cases except for Katz’s Back-Off the performance remains unchanged from the approach without interpolation. The regression caused by Katz’s Back-Off has the same root cause as described in the previous paragraph.

**Experiment 5.8: DBI Accuracy with Interpolation and Smoothing using  $\gamma = 71$**  Similarly, we repeat our experiment using a static  $\gamma = 71$ . As per Table 5.3, this does not improve performance over the baseline performance. However, we do see superior performance to fixed-weight Bayesian inference. Additionally, we observe that all interpolation method except for Katz’s Back-Off do not change the accuracy compared to the case without interpolation. This finding matches our finding for fixed-weight interpolation, indicating that interpolation techniques are not effective in our approach. We hypothesize that dynamic Bayesian inference outperforms fixed-weight Bayesian inference as a result of a lower average weight used in the dynamic weighting method (Figure 5.3). The lower weight causes the hybrid model to lean more towards the LLM, which in turn biases the accuracy of the hybrid model towards the baseline accuracy.

Overall, we conclude that interpolation and smoothing techniques are not effective in our approach; while the upper-bound performance shows slight improvements, we are unable to reach improvements beyond the baseline in practice.

**RQ1:** In conclusion, we find that hybrid models using Bayesian inference or dynamic Bayesian inference are not effective at improving the performance beyond the baseline. This indicates that the local n-gram model is no better at capturing localness than the global LLM, despite the n-gram accessing all project files while the LLM only accesses the current file. Additionally, we find that interpolation and smoothing techniques are not effective in our approach.

### 5.1.6. Shortcomings of N-gram Models

The performance of our hybrid LLM-n-gram models clearly indicates that local n-gram models cannot substantially improve the performance of global LLM models in traditional code completion tasks. Despite n-gram models being relatively adequate on their own, they are outclassed by LLMs to such an extent that combining them in a hybrid model causes overall regression rather than improvements. A key characteristic of LLMs is their ability to leverage large inputs to generate outputs that fit within a large context. We conclude that this ability is superior to the ability of the local n-gram model to use



---

cross-file contexts.

# 6

## Targeted Identifier Completion

Having established that local n-gram models cannot improve code completion accuracy in traditional code completion settings (i.e., applying local n-gram models for every prediction), we now investigate whether which parts of code are most likely to benefit from a local n-gram model. LLMs are great at writing syntactically correct code but typically struggle with the use of non-existent variables and functions. As such, applying a local n-gram model to only the parts of code that rely on the project context could lead to improved performance. Additionally, we investigate techniques beyond Bayesian inference to improve the hybrid global-local model, and explore whether it is possible to detect when a local n-gram model may be able to help the global LLM.

### 6.1. Shortcomings of LLMs

To understand how best to improve LLMs for code completion, it is imperative to first understand the limitations of the current state-of-the-art. The main challenge code LLMs face is knowledge of the relevant code context. LLMs are often trained on single functions or single files, restricting these models' abilities to situations where in-file context is sufficient to produce a correct prediction. This works relatively well, as the vast majority of code edits are made to existing files with imports and implementations that can be leveraged by the code model. However, this is not the case when writing new code or when writing code that is dissimilar to the code that is already present in the current file. In such cases, the model must be able to retrieve relevant code snippets to enhance the context the model is able to process. Many methods have been proposed to address this issue, ranging from probabilistic methods to semantic search. The main contribution of the additional context is typically to improve the model's knowledge of the current project, e.g., which classes, functions, and constants are available, and what naming conventions are used. Other details, such as code style, syntax, and the order of declarations, do not require as much additional context; usually, a single file is sufficient to inform the model of the programming language (thus syntax) and the preferred code style. For this reason it makes sense to develop methods that specifically target implementation-specific details (i.e., classes, functions, and variables) rather than language-specific details.

#### 6.1.1. Applicability of N-gram Models

The hybrid approach presented in Chapter 5 employed an hybrid LLM-n-gram model for every prediction; the n-gram model was able to influence syntactic elements of the output despite not being nearly as proficient at predicting general syntax as the LLM. Conversely, the LLM was able to impact context-sensitive details despite the n-gram model possibly being more proficient at such parts of the code. Rather than using both models for all elements of code, it could be beneficial to conditionally apply the n-gram model to only the parts of the code that are most likely to benefit from it. For this reason we investigate whether we can improve code completion performance by using a global-local hybrid where the local n-gram model is solely activated when predicting identifiers or function invocations. We specifically focus on identifiers and function invocations as these two offer a good balance between specificity and robustness. Targeting identifiers leads to many n-gram invocations, which may

`[_a-zA-Z][_a-zA-Z0-9]*`

**Figure 6.1:** Regular expression for identifying Identifiers

FunctionInvocation →	NestedAccessor ParenthesesExpression
NestedAccessor →	Name
NestedAccessor →	Name NestedAccessorInner
NestedAccessorInner →	Dot NestedAccessor
NestedAccessorInner →	BracketExpression
NestedAccessorInner →	BracketExpression NestedAccessorInner
Dot →	.
Name →	AlphaUnderscore NameInner
NameInner →	AlphanumericUnderscore
AlphaUnderscore →	AlphaUnderscoreInner AlphaUnderscore
AlphaUnderscore →	AlphaUnderscoreInner
AlphaUnderscoreInner →	_   a   b   c   ...   X   Y   Z
AlphanumericUnderscore →	AlphanumericUnderscoreInner AlphanumericUnderscore
AlphanumericUnderscore →	AlphanumericUnderscoreInner
AlphanumericUnderscoreInner →	0   1   2   ...   9   AlphaUnderscoreInner
BracketExpression →	[ BracketExpression ]
BracketExpression →	[ Token ]
BracketExpression →	[ ]
ParenthesesExpression →	( ParenthesesExpression )
ParenthesesExpression →	( Token )
ParenthesesExpression →	( )

**Figure 6.2:** Function Invocation Grammar

lead to over-application of the n-gram model and thus a higher false positive rate. Function invocations are longer and more specific, which could lead to fewer invocations with fewer false positives.

### Identifiers

We define identifiers using the regular expression displayed in Figure 6.1. This regular expression matches any sequence of alphanumeric characters and underscores, as long as the sequence does not start with a numeric character. We do not actively exclude keywords, as there is no reason to exclude these. While the LLM is likely able to perfectly predict keywords already, including them in the n-gram model does not decrease performance. This regular expression works on a broad range of programming languages, including Python, Java, C++, and JavaScript.

### Function Invocations

We construct a simple recursive parser to extract function invocations from the code. The parser is based on the grammar highlighted in Figure 6.2. This grammar supports invocations on nested accessors, i.e., syntax such as `a.b.c()`, where a function is invoked on some nested data structure or when class methods are invoked.

We apply a simple tokenization strategy before parsing. This strategy adds spaces around special symbols, after which it splits into tokens on single spaces. Special symbols are defined using the regular expression displayed in Figure 6.3.

As is the case for identifiers, this definition of a function invocation is language-agnostic and will work

[<sup>^</sup>A-Za-z0-9\_]

**Figure 6.3:** Regular expression for tokenizing Code

magic number = other number + another number

**Figure 6.4:** Identifier Tokens (Top) versus LLM Tokenizer Tokens (Bottom)

for Python, Java, C++, JavaScript, and numerous other languages. Additionally, the implementation of the parser allows for partial parsing, i.e., it detects when the input code ends before the end of a function invocation. Note that this parser is not perfect, but suitable for our use case.

## 6.2. Targeted LLM steering

Having explored Bayesian inference and its shortcomings, we now turn to improving code completion by steering LLMs in the right direction using specific features of code such as identifiers and function invocations, rather than using all code tokens.

### 6.2.1. Identifiers and Function Invocations

#### Aligning Tokens

To be able to use identifiers and function invocations in the context of our global-local hybrid, we must ensure that we represent them using the same tokens as our hybrid model. When extracting identifiers and function invocations this clearly is not the case; identifiers can be any sequence of characters, while function invocations have their own tokenization scheme. Thus, we process tokens to ensure we can create n-gram models based on identifiers and function invocations.

Figure 6.4 clearly displays the issue. The tokenizer used by the LLM typically includes leading spaces, and splits up long words into multiple tokens. This leads to overlapping tokens when comparing LLM tokens to identifiers (or the tokens used by the function invocation parser).

**Identifiers** Addressing this issue is relatively straightforward, namely, we first tokenize the code using the LLM tokenizer, and subsequently use a sliding window to extract identifiers. As larger identifiers can be created by simply adding more alphanumeric or underscore characters to an existing identifier, we keep expanding the sliding window until we can no longer do so. After growing the window to the largest possible size, we store the tokens that belong to the identifier, after which we start a new sliding window at the next index. Since the tokenizer includes leading spaces, we adjust the identifier extraction to allow for leading whitespaces.

**Function Invocations** Function invocations can be extracted in a similar sliding window approach. However, now we grow the sliding window as long as we are able to parse a partial function invocation. The moment we find a full function invocation or are unable to parse a partial function invocation, we stop growing the sliding window. In the former case, we store the tokens that belong to the function invocation. In either case, we start a new sliding window at the next index.

The fact that our function invocation parser supports partial parsing considerably reduces the computational overhead of function invocation extraction.

#### Training

As function invocations and identifiers have a clear end, we must ensure that our n-gram model supports early termination. Our original n-gram model simply stores all subsequences of length  $n$  to the n-gram model. This in turn causes the model to be able to continue predicting tokens indefinitely as long as the input occurs in the training data. However, as identifiers and function invocations are never immediately followed by another identifier or function invocation, we should relinquish control to the LLM whenever the identifier or function invocation is ends. To achieve this, we introduce a special `<EOS>` token that is added at the end of every identifier and function invocation in the training data. The probability mass of

Experiment	Accuracy
Baseline	75.90
BI-UpperBound	83.13
DBI-UpperBound	83.13
$w = 0.5$	77.11
$\gamma = 51$	75.90

**Table 6.1:** BI and DBI Hybrid Model Accuracy (Identifiers)

the `<EOS>` token is distributed to the LLM if it is encountered by the n-gram model during inference, as the n-gram model is not responsible for predicting tokens that are not identifiers or function invocations.

Since our n-gram model still has a fixed sequence length  $n$ , we pad using the `<EOS>` token when subsequences have a length below  $n$ . This ensures that the n-gram model is able to create predictions at the end of identifiers. For instance, for some identifier `mynumber` consisting of LLM tokens `[my, number]`, we will add the following subsequences in an n-gram model with  $n = 3$ :

- `[my, number, <EOS>]`
- `[number, <EOS>, <EOS>]`

#### Dataset Creation

We once more create datasets based on the Long Code Arena Project-Level Code Completion benchmark [43]. We create two datasets; one targeting identifiers and one targeting function invocations. The two datasets are constructed similarly to the previous dataset, except that instead of invoking code completion after the first non-whitespace token of a line, we invoke it after the first identifier or function invocation token using the approach highlighted in subsection 6.2.1.

#### Inference

Using an n-gram model to predict identifiers and function invocations requires a slight change to inference. Previously, we could simply use the last  $n$  tokens of the input to lookup the next-token probability distribution. However, since we discarded non-identifiers/function-invocations during training, we should do the same during inference. To this end, we extract the last  $k < n$  tokens of the input, given that they form a (partial) identifier or function invocation. This subsequence can then be used as input for inference as before.

Similar to the way we extract identifiers and function invocations during training (see subsection 6.2.1), we iterate over the input tokens to extract the longest possible (partial) identifier or function invocation. Rather than using a sliding window, we now fix the end of the window to the end of the input and only grow the window in the leftward direction. This ensures that we capture the part of the identifier or function invocation that has already been written. If this sequence has a length above  $n$ , we truncate it to  $n$  tokens by discarding the first  $n - k$  tokens.

### 6.2.2. Targeted Bayesian Inference

We investigate whether a global-LLM-local-n-gram hybrid improves over the baseline when targeting specifically identifiers and function invocations. We use the same approach as in Chapter 5 and determine the upper bound performance, and the performance when fixing  $w_{ngram}$  and  $\gamma$ .

#### Targeting Identifiers

We first investigate to what extent a local n-gram model trained on identifiers can improve code completion performance. As our definition of identifiers is quite broad, the n-gram model is frequently applied, which could lead to higher theoretical performance gains than function invocations as these are less frequently found in code (while function invocations are still common, every function invocation will consist of at least one identifier. As most functions accept multiple arguments, the number of identifiers will typically dominate the number of function invocations present in code).

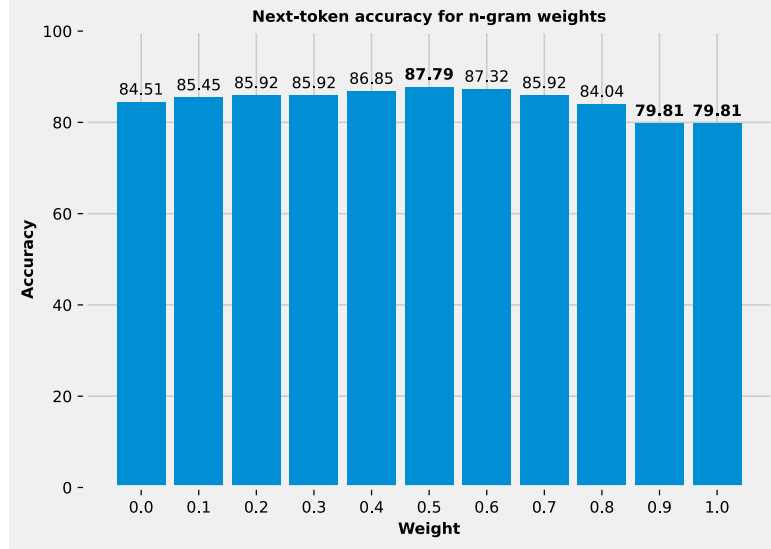


Figure 6.5: BI Hybrid Model Next-token Accuracy (Identifiers)

**Experiment 6.1: Baseline LLM Code Completion (Identifiers)** As shown in Table 6.1, the baseline accuracy is 42.11%. The baseline next-token accuracy comes in higher at 84.51%. The observed gap between accuracy and next-token accuracy is caused by the fact that the ground truth over which the accuracy is computed contains more than a single token, which is not the case for next-token accuracy. The next-token accuracy is slightly higher than the baseline next-token accuracy when applying a hybrid modal on all parts of code, which resulted in a next-token accuracy of 81.67%. This indicates that identifiers are slightly easier to predict than other parts of the code. We attribute this to the fact that given some prefix and knowledge of naming conventions in the current file, it is possible to continue existing identifiers with relatively high certainty. This leads to higher next-token accuracies as many identifiers consist of multiple tokens.

**Experiment 6.2: UB on BI Accuracy (Identifiers)** There is a clear difference of 7.23%pt between the baseline accuracy and upper bound accuracy. This is a far larger difference than the difference observed in subsection 5.1.3 (2.1%pt). The large difference indicates a potential for concrete performance gains. The next-token accuracy for different weights  $w_{ngram}$  is displayed in Figure 6.5. While  $w_{ngram} = 0.5$  performs best on average, even  $w_{ngram} = 1.0$  (i.e., only using the n-gram model) leads to an outperformance over the baseline. This clearly shows that using the local n-gram model for specific elements of code can lead to improved performance.

**Experiment 6.3: BI Accuracy with  $w_{ngram} = 0.5$  (Identifiers)** We select  $w_{ngram} = 0.5$  for this experiment, as this configuration worked best on average based on Figure 6.5. This weight leads to an accuracy of 77.11%, which improves over the baseline accuracy (75.90%) by 1.21%pt. This is a large change from the previous experiments that did not specifically focus on identifiers. Previously, the local n-gram model caused performance regression, whereas we now observe an improvement in performance as a direct result of the n-gram model.

**Experiment 6.4: UB on DBI Accuracy (Identifiers)** The theoretical upper bound for dynamic Bayesian inference matches that of fixed-weight Bayesian inference, at 83.13%. We do, however, observe lower next-token accuracy for dynamic Bayesian inference, as shown in Figure 6.6. This finding may indicate that this specific dynamic Bayesian inference technique is not effective, e.g., due to its assumption that n-gram frequencies translate well to the likelihood of the n-gram model being correct. These findings are consistent with the results of subsection 5.1.3, as in both cases fixed-weight Bayesian inference achieved a higher next-token accuracy.

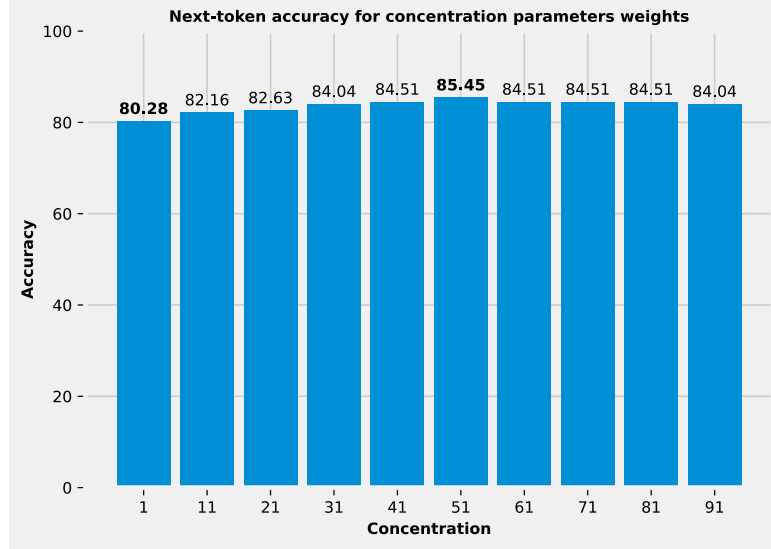


Figure 6.6: DBI Hybrid Model Next-token Accuracy (Identifiers)

Experiment	Accuracy
Baseline	40.38
BI-UpperBound	40.38
DBI-UpperBound	40.38

Table 6.2: BI and DBI Hybrid Model Accuracy (Function Invocations)

**Experiment 6.5: DBI Accuracy with  $\gamma = 51$  (Identifiers)** As per Table 6.1, we set  $\gamma = 51$  for this experiment. This configuration leads to an accuracy of 75.90%, which exactly matches the baseline performance. This outcome matches the outcome of subsection 5.1.3, where dynamic Bayesian inference was also unable to beat the baseline performance. This reinforces that the dynamic approach may not be effective.

Overall, it is clear that targeting identifiers can lead to improved performance over the baseline LLM. Our experiments verify the hypothesis that the local n-gram model is not effective at predicting language-specific elements of code such as syntax. Additionally, we observe that the performance when using dynamic Bayesian inference is worse than when using fixed-weight Bayesian inference.

#### Targeting Function Invocations

Knowing that local n-gram models trained on identifiers can be effective, we continue by investigating whether focusing function invocations can have a similar effect. Function invocations are not as common as identifiers, but are generally longer which could result in fewer n-gram invocations, but at a higher precision.

**Experiment 6.6: Baseline LLM Code Completion (Function Invocations)** As shown in Table 6.2, the baseline accuracy for function invocations is 40.38%. The baseline next-token accuracy is 83.42%, which is very similar to the next-token accuracy for identifiers.

**Experiment 6.7: UB on BI Accuracy (Function Invocations)** The upper bound performance for the hybrid model trained on function invocations is identical to the baseline accuracy (40.38%). This shows that there is no room to improve over the baseline using Bayesian inference. Additionally, Figure 6.7 shows that the next-token accuracy is at its peak when not applying the local n-gram model whatsoever ( $w_{ngram} = 0$ ). We also observe that the next-token accuracy with  $w_{ngram} = 1.0$  is much lower compared to the same scenario for the *identifier* n-gram model. Based on this, it is clear that tokens inside function invocations are far less predictable based on local context than identifiers. We attribute this to the fact



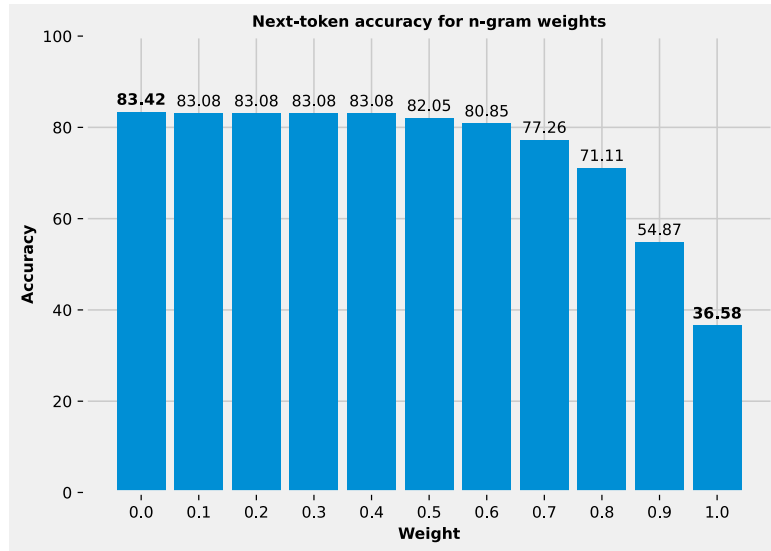


Figure 6.7: BI Hybrid Model Next-token Accuracy (Function Invocations)

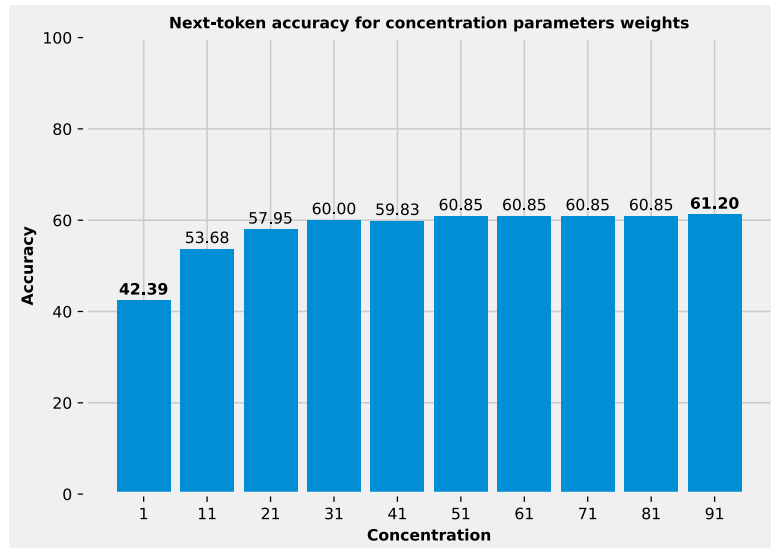


Figure 6.8: DBI Hybrid Model Next-token Accuracy (Function Invocations)

that function invocations are far larger and not likely to be the same every time; a single change, whether it be a argument name or literal, results in the n-gram model being unable to predict the next token. Another factor is the fact that the sequences the n-gram model stores are limited in length, which can quickly cause the model to forget which function is being invoked.

**Experiment 6.8: UB on DBI Accuracy (Function Invocations)** We observe the same results for dynamic Bayesian inference; it is unable to improve performance beyond the accuracy of the baseline. Figure 6.8 affirms that tokens within a function invocation are more challenging to predict than identifier tokens.

Overall, we conclude that raw function invocations are not suitable for our use case. While we believe that code completion for function invocations can likely be improved using a different approach, the forgetfulness and specificity of n-gram models make it difficult to predict function invocations without additional processing.

Experiment	Accuracy
Baseline	75.90
Top-k-UB	84.34
Top-p-UB	84.34
$k = 19$	77.11
$p = 0.9$	73.49

**Table 6.3:** Top-k/p Masking Hybrid Model Accuracy (Identifiers)

**RQ2:** To conclude, we find that targeting solely identifiers is a viable approach to adapting code completion to new codebases. While function invocations can also be informative, the dynamic nature of function arguments and the low context window size of our n-gram model make it impossible to improve code completion performance by targeting function invocations. Nevertheless, our success in targeting identifiers is a strong indication that local n-gram models can be of use in cooperation with a global LLM.

### 6.2.3. N-gram Masking Techniques

Having explored using Bayesian inference to improve code completion, we now turn our focus to alternative approaches to close the gap between theoretical accuracy gains and actual performance gains. The sole difference between the previous upper bounds and actual accuracy is the fact that the upper bound experiments used an oracle to select the optimal  $w_{ngram}$  and  $\gamma$  values. Finding the best weight value is a non-trivial task: the optimum weight is dependent on both the LLM probability distribution and the n-gram probability distribution. These are both in turn dependent on the training data and training objectives of the two models. As patterns within probability distributions are inconsistent between different LLMs, inputs, and project contexts, we deem it very challenging to find the optimal weight value. As the dynamic weighting approach used by Tu, Su, and Devanbu did not result in improvements, we instead focus on eliminating the need to be able to predict the best weight value. We achieve this through masking techniques; rather than combining probability distributions, we instead use the n-gram model's output distribution to select which tokens the LLM is able to assign probability mass to. This may further push the theoretical performance improvements and close the gap between theoretical and practical improvements. This approach gives the LLM the responsibility of correctly ranking tokens, while the n-gram model is responsible for selecting which tokens to assign probability mass to. As LLMs excel at writing syntactically correct code but often struggle with the use of non-existent variables, it makes sense for them to rank tokens rather than select them. Conversely, the n-gram model has been trained to know which tokens are used within the project context, making it perfect for this task. In this approach we call the n-gram model the *retriever*, and the LLM the *ranker*.

Based on our findings from subsection 6.2.2 we perform our experiments using local n-gram models trained on identifiers for the following experiments, i.e., Targeted Identifier Completion (TIC).

#### Top-k Masking

**Experiment 6.9: UB on Top-k Masking Accuracy (Identifiers)** To establish whether a retriever-ranker system can improve performance, we first investigate the theoretical upper bound performance. We use a similar approach to previous experiments by using an oracle for  $k$ , after which we use the LLM to rank the top-k tokens in the n-gram model's output distribution. Additionally, we compute the precision and recall for different values of  $k$ , allowing us to make an informed decision when setting a value for  $k$ . This results in a theoretical upper bound accuracy of 84.34% (Table 6.3), exceeding the previous best upper bound of 83.13% by 1.21%pt, and exceeding the baseline performance of 75.90% by 8.44%pt.

Figure 6.9 displays the next-token accuracy for different values of  $k$ . This figure shows that the next-token is typically within the top-19 n-gram tokens, as the next-token accuracy converges at this value for  $k$ . Despite  $k$  growing larger beyond this point the next-token accuracy remains constant, indicating that the LLM is able to make sensible decisions even when given many options for possible next tokens. This indicates that it is likely favorable to prioritize recall over precision when selecting  $k$ . We believe that this is a result of the n-gram model ruling out a large part of tokens that would otherwise have

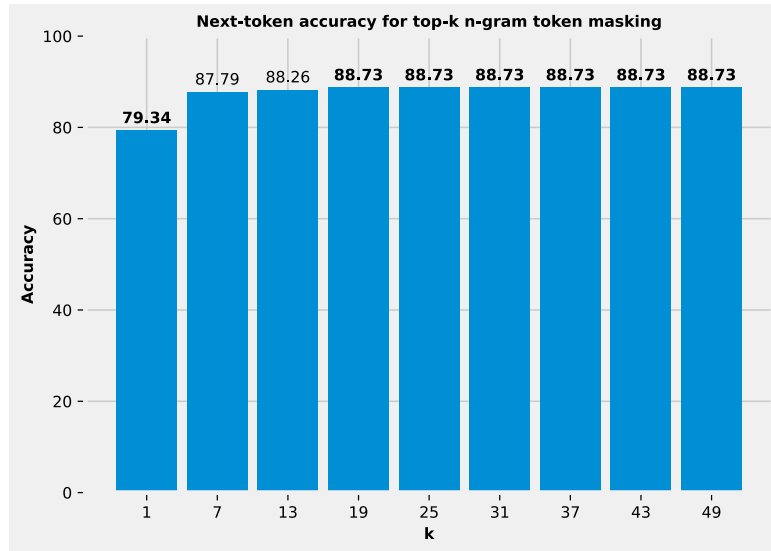


Figure 6.9: Top-k Masking Hybrid Model Next-token Accuracy (Identifiers)

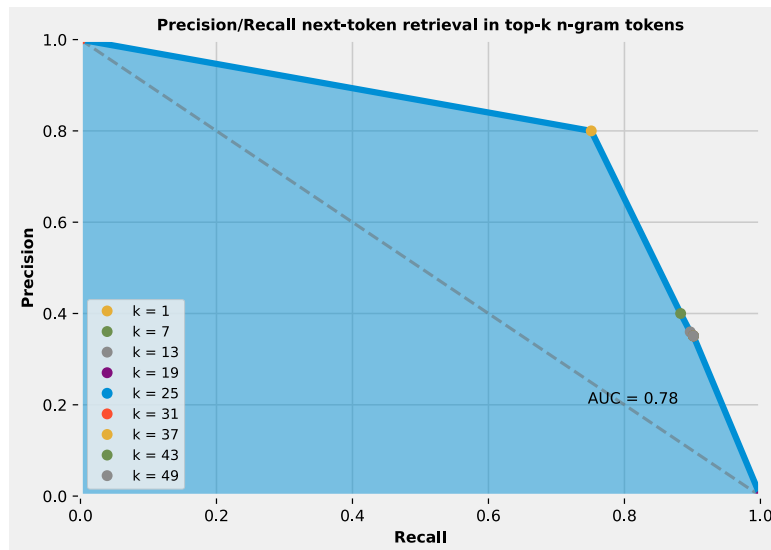
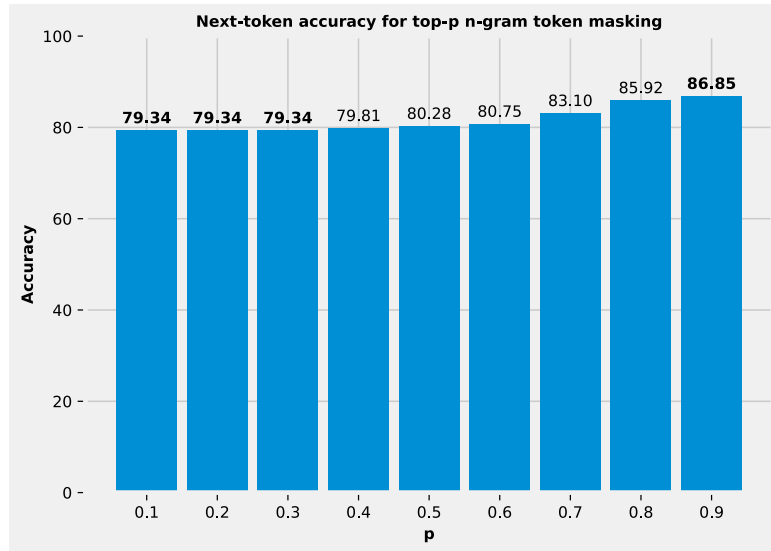


Figure 6.10: Top-k Masking Precision and Recall (Identifiers)

been relatively likely. Such tokens are sensible but simply do not fit within the project context, leading to them not being present in the n-gram's list of known tokens.

The highest achieved next-token accuracy is 88.73%, which improves upon the hybrid model applying Bayesian inference on identifiers (84.51%). Consequently, the next-token accuracy is also higher than the baseline targeting all parts of code (81.67%), indicating that targeting identifiers can lead to higher accuracy.

Figure 6.10 displays the precision and recall for different values of  $k$ . The convergence observed in Figure 6.9 is also observed in this figure.  $k = 1$  is relatively isolated, showing a balance between precision and recall. Other values for  $k$  are more closely distributed and tend towards a higher recall in exchange for a lower precision. The area under curve (AUC) is 0.78, indicating that the n-gram model frequently includes identifier tokens that should be predicted, but does not always contain them. This shows that the local n-gram model cannot always help the global LLM, e.g., when writing code with variable names and functions that have never been used before. We did not perform any filtering of test samples, so we believe that this area under curve displays that distribution of context-sensitive versus non-context-sensitive identifier completions does allow for improvements by the identifier local n-gram



**Figure 6.11:** Top-p Masking Hybrid Model Next-token Accuracy (Identifiers)

in practice.

Based on this experiment, it is clear that a masking approach may present a viable improvement over the bayesian-inference-based hybrid model. Rather than controlling the weight, we can now control precision and recall, allowing us to weigh our concerns to determine a value that fits our use case. Such configuration was not possible for a Bayesian inference approach, showing another benefit of a masking method. The greatly increased upper bound accuracy indicates that masking may be superior to Bayesian inference.

**Experiment 6.10: Top-k Masking Accuracy with  $k = 19$  (Identifiers)** Based on Figure 6.14 we set  $k = 19$  for this experiment. This gives us an accuracy of 77.11% (Table 6.3), which is an improvement of 1.21%pt over to the baseline but matches the Bayesian inference approach. This shows that although the masking approach is able to achieve a higher theoretical accuracy than Bayesian inference, it does not outperform Bayesian inference in practice. This result indicates that making the LLM responsible for ranking tokens selected by the n-gram model is not a perfect approach; it is likely that incorporating knowledge of the project context into the ranking process could improve results (see Section 6.2).

#### Top-p Masking

As an alternative to top-k masking, we consider top-p masking. Top-p masking is similar to top-k masking but uses a dynamic value for  $k$  depending on the total probability mass of the top-k tokens.  $k$  is chosen such that it is the minimum quantity for which the sum of the top-k token probabilities is greater than or equal to  $p$ .

**Experiment 6.11: UB on Top-p Masking Accuracy (Identifiers)** As per Table 6.3, we find that top-p masking achieves a theoretical upper bound of 84.34%, matching upper bound of top-k masking and thus outperforming the baseline.

Nevertheless, considering the next-token accuracy (Figure 6.11) shows that top-p masking performs worse than top-k masking on a single-token granularity level. The fact that the two achieve the same accuracy is likely due to increased single-token mistakes in test samples that our model already made a mistake in; this does not affect the accuracy but does decrease the next-token accuracy. Differently from top-k masking, we do not observe early convergence. This indicates that the probability mass of the n-gram model is likely concentrated in the most likely tokens.

We plot the average number of tokens for different values of  $p$  in Figure 6.12. This plot clearly confirms our hypothesis; when  $p = 0.9$  we observe an average token count of 1.96. We attribute this unnatural distribution of probability mass to the fact that our local n-gram model has a very limited effective

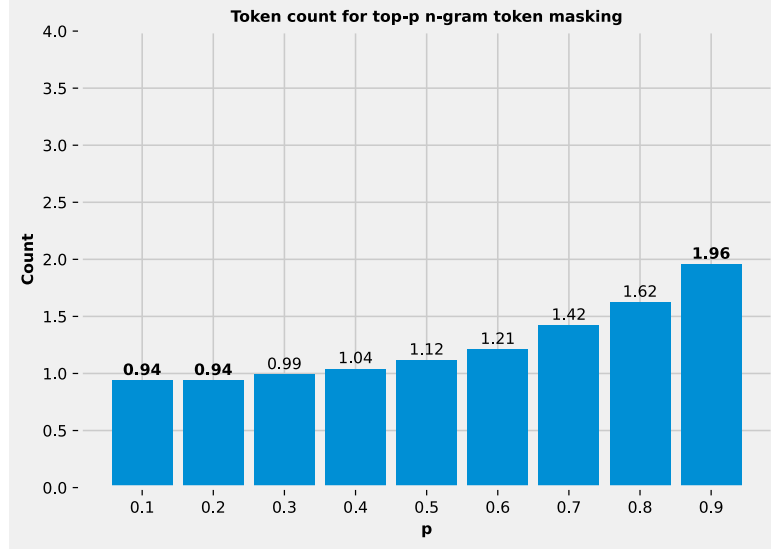


Figure 6.12: Top-p Masking Average Number of Tokens (Identifiers)

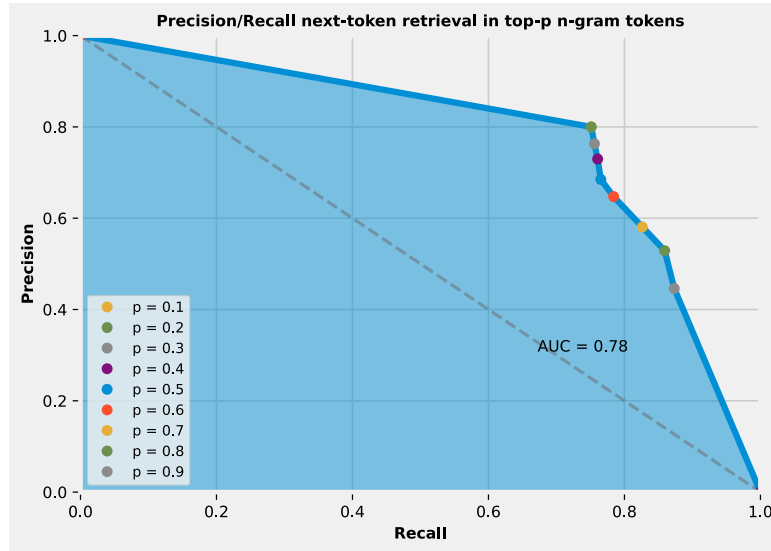


Figure 6.13: Top-p Masking Precision and Recall (Identifiers)

vocabulary due to the precise nature of the identifiers we capture. E.g., when predicting an identifier given some prefix, there may be just one known continuation. In this case all probability mass will be given to one token, leading to just one token being selected despite setting  $p = 0.9$ . As  $p = 0.9$  results in an average token count of 1.96 it is not always the case that there is just one known continuation, but likely is a very common scenario. Nevertheless, it is clear that the precise nature of the local n-gram model trained on identifiers makes top-p masking less effective.

For completeness, we also investigate the precision, recall, and area under curve for top-p masking. As shown in Figure 6.13, the precision and recall for different values of  $p$  are similar to those for top-k masking. As to be expected based on the token counts for different  $p$  values, we notice that a very similar precision-recall can be achieved as  $k = 1$  for top-k masking. However, we do recognize that top-p masking can achieve different precision-recall balances than top-k masking. Nevertheless, top-p masking cannot achieve a recall that matches top-k masking due to its low average token count. This explains the lower next-token accuracy, as the LLM was previously shown to favor high recall over high precision. Unsurprisingly, we observe an identical AUC of 0.78 for top-p masking as for top-k masking.

Experiment	Accuracy
Baseline	75.90
Top-k-UB	75.90
Top-p-UB	75.90
Tie-Breaking-UB	77.11

**Table 6.4:** Top-k/p/Tie-Breaking Masking Hybrid Model Accuracy (Identifiers, LLM Retriever)

**Experiment 6.12: Top-p Masking Accuracy with  $p = 0.9$  (Identifiers)** To verify the previous findings, we repeat our experiments with  $p = 0.9$  for top-p masking. This results in an accuracy of 73.49% (Table 6.3), confirming that top-p masking is not a suitable approach.

Overall, we conclude that top-p masking is not an appropriate approach for our use case because the n-gram model often has few tokens that make up the majority of the probability mass.

**RQ3.1:** To conclude, we find that TIC with top-k and top-p masking pushes the upper bound on theoretical accuracy higher than Bayesian inference did. When fixing parameters we observe equal accuracy and improved next-token accuracy for top-k masking while top-p masking achieves lower accuracy and next-token accuracy. The latter is attributed to the fact that our local n-gram model has a high concentration of probability mass in the most likely tokens. Our findings indicate that top-k may be a viable approach, matching or beating Bayesian inference in practice.

#### 6.2.4. LLM Retrievers

As mentioned previously, using an LLM as ranker may not be the best approach as the LLM has limited knowledge of the project context. In this section, we explore whether the opposite approach is viable, i.e., using an LLM retriever and an n-gram ranker. We initially chose to use the n-gram model as ranker as we hypothesized that project context is most important for token selection, however, the opposite approach may work too if the distribution of LLM training data is such that tokens that fit within a varied range of contexts are all assigned relative high probabilities. Additionally, while the top-p method does not work well with n-gram models, it may work better when applied with LLM retrievers. We repeat our previous top-k and top-p masking experiments and explore a different *tie-breaking* masking technique. The aim of these experiments is to determine to what extent the different components of our masking approach are dependent on project context. Once again, our experiments use a hybrid model focused on identifiers.

##### Top-k Masking

**Experiment 6.13: UB on Top-k Masking Accuracy (Identifiers, LLM Retriever)** We start by establishing the upper-bound performance of top-k masking. We find the upper bound on accuracy to be 75.90%, matching the baseline performance. This indicates that using the top-k LLM tokens is not a viable approach to improving code completion performance, and that using the LLM as retriever does not seem to be a better option than using the n-gram for this task.

For completeness, we display the next-token accuracy for different values of  $k$  in Figure 6.14. We observe that the next-token accuracy is consistently 84.51% for all values of  $k$ , matching the baseline next-token accuracy. This indicates that this hybrid model always uses the most probable token, and that using the n-gram model to rank tokens does not seem to be a viable option. We explain this unexpected behavior by the fact that in most situations, the n-gram model cannot rank tokens as its effective vocabulary is much smaller than that of the LLM. I.e., the n-gram model can only rank tokens when it knows the prefix and can only assign probabilities to tokens that it has seen following the prefix. This leads to very few scenarios where the n-gram model can rank tokens, leading to our model falling back to the original LLM token probabilities. While this does not always happen, in the cases where the n-gram model does rank tokens, it agrees

As the LLM predicts probabilities for all tokens in its vocabulary, we expect a higher maximum recall. The precision-recall curve displayed in Figure 6.15 confirms this finding. We observe that for  $k = 1$  we can achieve a good precision-recall balance, but that for all other values of  $k$  the recall becomes very high at the cost of a very low precision. While this leads to a higher area under curve than our

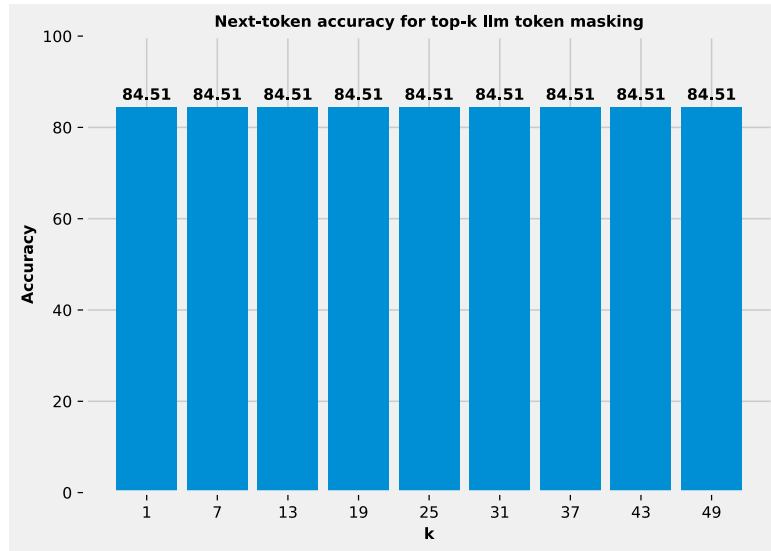


Figure 6.14: Top-k Masking Hybrid Model Next-token Accuracy (Identifiers, LLM Retriever)

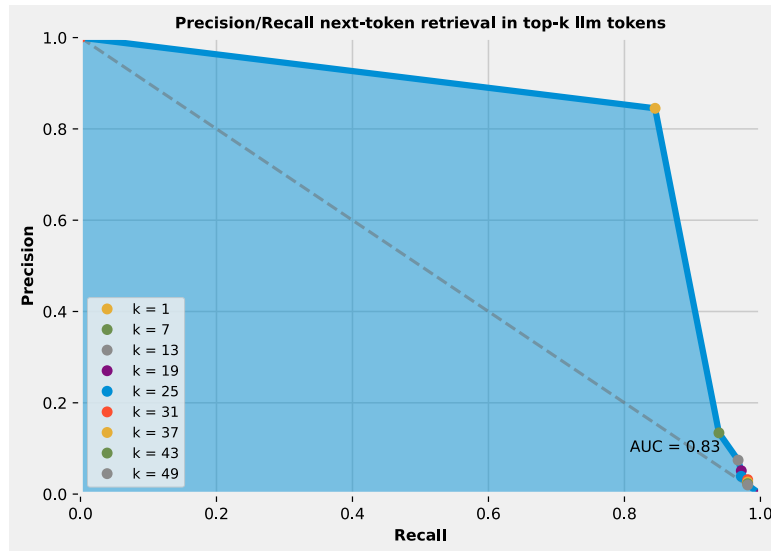


Figure 6.15: Top-k Masking Precision and Recall (Identifiers, LLM Retriever)

hybrid model that uses the n-gram model to select the top-k tokens, this is not necessarily better. We previously found that using  $k = 19$  created a good precision-recall balance, but this is no longer the case in this plot as it heavily favors recall. This leaves  $k = 1$  as the only viable option, but as this matches the baseline's greedy decoding, using the LLM to select tokens does not help the hybrid model perform better than the baseline.

### Top-p Masking

**Experiment 6.14: UB on Top-p Masking Accuracy (Identifiers, LLM Retriever)** Based on the results of top-k masking, it appears unlikely for top-p masking to be effective. Nevertheless, we investigate top-p masking to determine how it compares to top-k masking. Running our model to determine the upper-bound performance, we are able to achieve 75.90% accuracy, matching the baseline and top-k performance.

As shown in Figure 6.16, the next-token accuracy also matches the baseline and top-k performance.

Lastly, the precision-recall curve for top-p masking (Figure 6.17) is very similar to the top-k curve. Comparing the precision-recall curves for top-k-LLM and top-p-LLM to the curves for top-k-n-gram and top-

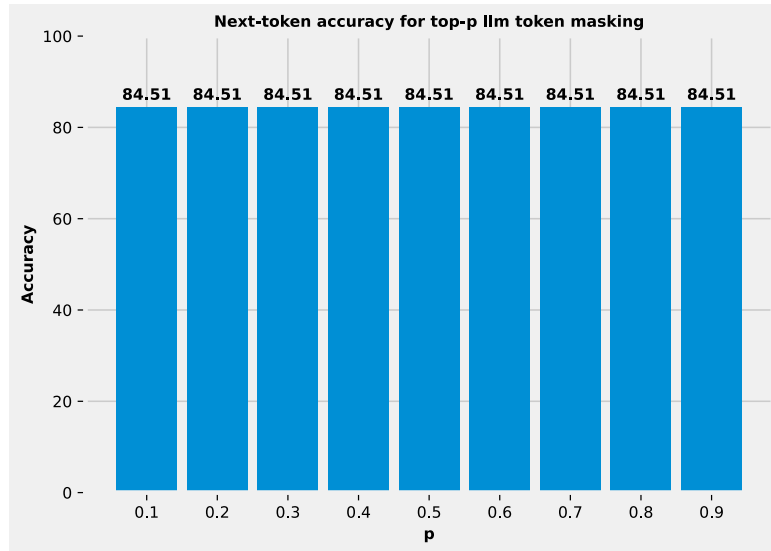


Figure 6.16: Top-p Masking Hybrid Model Next-token Accuracy (Identifiers, LLM Retriever)

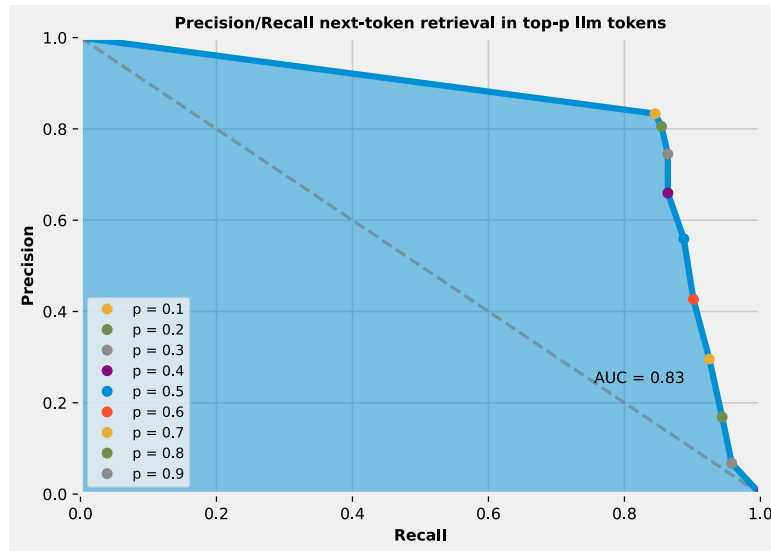


Figure 6.17: Top-p Masking Precision and Recall (Identifiers, LLM Retriever)

p-n-gram shows a similar trend where the top-p method leads to different parameter values leading to a more uniform distribution of precision-recall combinations. Top-k, on the other hand, quickly converges to high recall with low precision. This shows that similarly to the n-gram model, the LLM heavily concentrates its probability mass in the most likely tokens. Plotting the number of tokens selected for different values of  $p$  affirms this, as shown in Figure 6.18. While the number of tokens is higher than the number of tokens when using the n-gram model, the number of tokens is still very low unless  $p$  is set to a very high value.

Similar to top-k masking, using the LLM to select the top-p tokens is not a viable approach.

#### Tie-Breaking Masking

As an alternative to top-k and top-p masking, we investigate letting the n-gram model rank tokens that have probabilities not far from another. We first order all tokens based on their probability, after which we add a token to the list of tied tokens as long as their probability is larger than  $c$  multiplied by the probability of the previous token.  $c$  is a configurable parameter that determines how close probabilities must be to be deemed tied. This method is not suitable when using an n-gram model to select tokens,



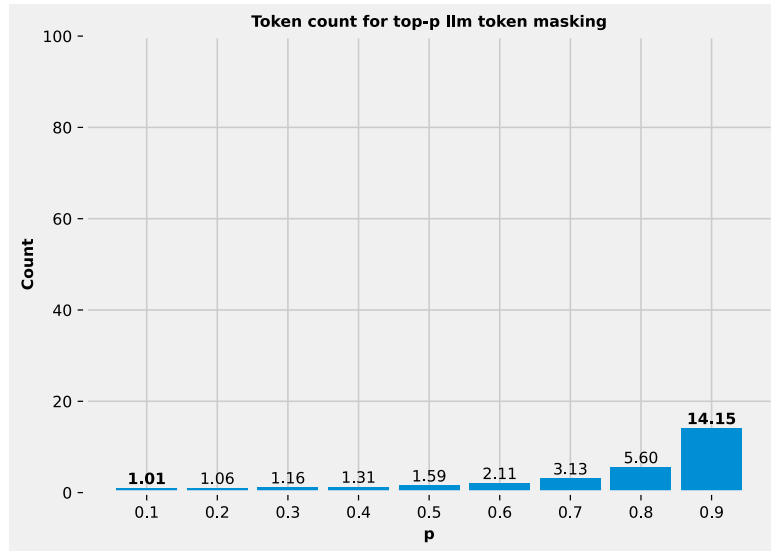


Figure 6.18: Top-p Masking Average Number of Tokens (Identifiers, LLM Retriever)

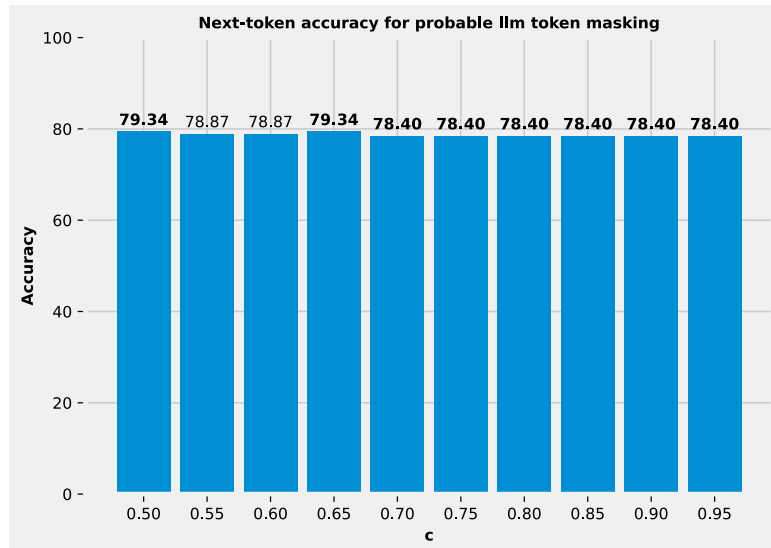


Figure 6.19: Tie-Breaking Masking Hybrid Model Next-token Accuracy (Identifiers, LLM Retriever)

as the output of the n-gram model is nearly always very sparse. This leads to many cases where there is just one possible token and no tie-breaking to be done.

**Experiment 6.15: UB on Tie-Breaking Masking Accuracy (Identifiers, LLM Retriever)** We start by determining the theoretical upper bound on accuracy. As per Table 6.4, we find that the upper bound performance is 77.11%. While this is an improvement over the baseline, and better than top-k and top-p, it is not a considerable improvement.

We display the next-token accuracy for different values of  $c$  in Figure 6.19. The next-token accuracy is substantially *lower* than that of previous experiments. The fact that the accuracy improved regardless is likely because the regression primarily impacts samples that the baseline model predicted incorrectly in the first place.

The precision-recall curve displayed in Figure 6.20 shows that only one specific precision-recall combination is possible. This combination is very similar to top-k with  $k = 1$ , showing that this selection strategy likely results in a low number of tokens in many cases.

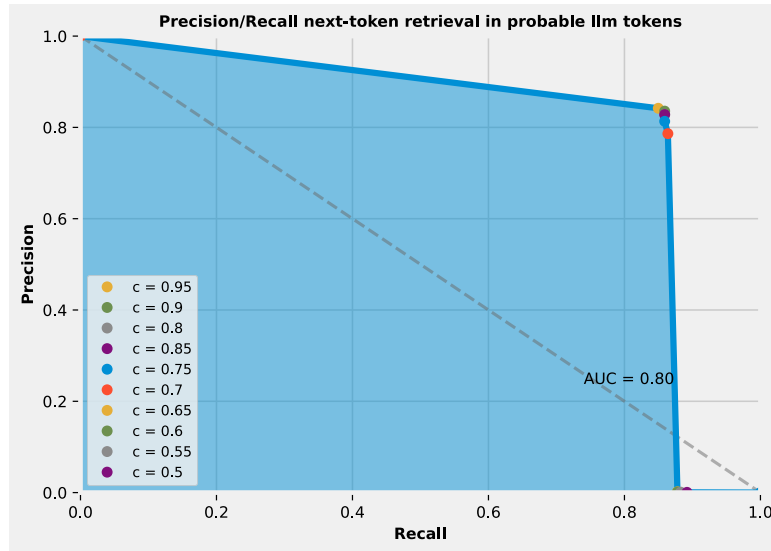


Figure 6.20: Tie-Breaking Masking Precision and Recall (Identifiers, LLM Retriever)

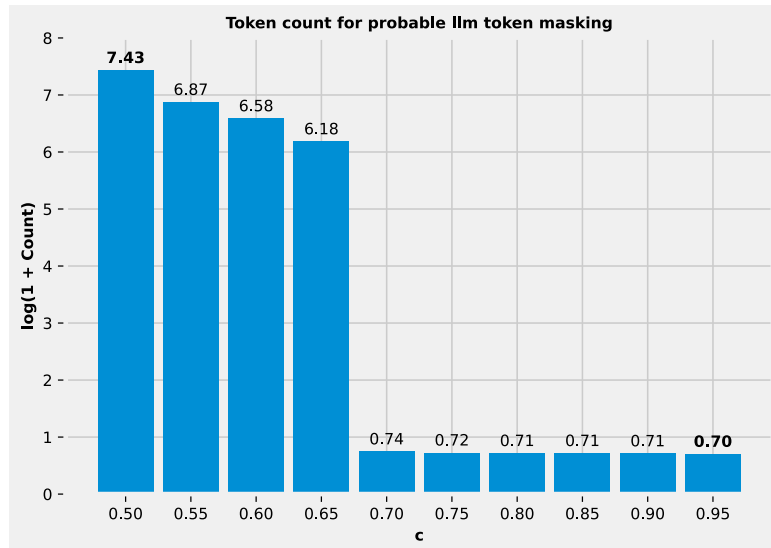


Figure 6.21: Tie-Breaking Masking Average Number of Tokens (Identifiers, LLM Retriever)

As expected, the number of tokens for different values of  $c$  is consistently low, as shown in Figure 6.21. For values  $c \geq 0.7$  we see an average token count of around 1, while values  $c < 0.7$  lead to an average token count over 1 000. This indicates that the tie-breaking strategy is not effective due to the skewed nature of the probability distribution.

Overall, we conclude that using the LLM to select tokens is not a feasible approach.

**RQ3.2:** To conclude, using LLMs as retrievers leads to worse performance than using n-gram models as retrievers. This appears to confirm our hypothesis that using knowledge of project context to constrain the list of tokens is superior to using knowledge of project context to rank tokens.

### 6.2.5. N-Gram-LLM Retriever-Ranker System

We have established that using n-gram retrievers with LLM rankers can improve code completion accuracy in context-sensitive code such as identifiers. Our masking approach completely discards n-gram probabilities after token selection despite Bayesian inference having improved performance. For this reason, we investigate whether accuracy can be further enhanced by performing Bayesian inference

Experiment	Accuracy
Baseline	75.90
Top-k-B-UB	84.34
Top-p-B-UB	84.34
$k = 19, w_{ngram} = 0.5$	78.31
$p = 0.9, w_{ngram} = 0.5$	74.70

**Table 6.5:** Top-k/p Masking + BI Hybrid Model Accuracy (Identifiers)

after the retriever-ranker system selects LLM probabilities. I.e., we continue by incorporating knowledge of project context in the ranker instead of just the retriever to establish to what extent Bayesian inference *after* masking can improve code completion.

#### Top-k with Bayesian Inference

We combine the approaches of previous experiments by performing Bayesian inference after using the retriever-ranker system to obtain a probability distribution.

**Experiment 6.16: UB on Top-k Masking + Bayesian Inference Accuracy (Identifiers)** As displayed in Table 6.5, the theoretical upper bound accuracy for top-k masking with Bayesian inference is 84.34%. This matches the upper bound accuracy for top-k masking without Bayesian inference.

Figure 6.22 displays the next-token accuracy for different values of  $k$  and  $w$ . The best next-token accuracy is achieved for  $w_{ngram} = 0.5$  and  $k \geq 19$  at 89.67%. This exceeds the next-token accuracy of top-k without Bayesian inference (88.73%) by a slight margin. This is also the case for Bayesian inference (87.79%), indicating leveraging both techniques can lead to further improvements. Additionally, we find that the optimal parameter values observed when applying both techniques simultaneously match the values that were found to be optimal in isolation. For all weights we observe eventual convergence as  $k$  increases, once more confirming that the LLM is a good ranker, even when the retriever selects too many tokens.

**Experiment 6.17: Top-k Masking + BI Accuracy with  $w_{ngram} = 0.5$  and  $k = 19$  (Identifiers)** We set  $w_{ngram} = 0.5$  and  $k = 19$  for this experiment, resulting in an accuracy of 78.31% (Table 6.5). This exceeds the accuracy of the baseline (75.90%), top-k masking (77.11%), and Bayesian inference (77.11%). These results show that the performance of LLMs on context-sensitive areas like identifiers can be improved through the use of retrieval techniques and Bayesian inference.

#### Top-p with Bayesian Inference

We repeat our experiments using top-p masking with Bayesian inference.

**Experiment 6.18: UB on Top-p Masking + BI Accuracy (Identifiers)** The highest possible accuracy for top-p masking with Bayesian inference is 84.34%, matching the upper bound for top-k.

Despite the equal upper bound accuracy, top-p masking with Bayesian inference has a lower next-token accuracy in general compared to top-k masking with Bayesian inference. As shown in Figure 6.23, the highest achievable next-token accuracy is 87.79%, matching Bayesian inference in isolation. Similar to Bayesian inference and top-k masking with Bayesian inference, we observe that  $w_{ngram} = 0.5$  appears to be the optimal value. Additionally, we observe that higher  $p$  values generally lead to higher next-token accuracies. This finding matches our previous findings for top-p masking and indicates that the probability mass is concentrated in the most likely few tokens of the n-gram model.

**Experiment 6.19: Top-p Masking + BI Accuracy with  $w_{ngram} = 0.5$  and  $p = 0.9$  (Identifiers)** We use  $w_{ngram} = 0.5$  and  $p = 0.9$  for this experiment, leading to an accuracy of 74.70% (Table 6.5). Similar to top-p masking in isolation, this is a regression compared to the baseline (75.90%). However, we observe that the regression is smaller when combined with Bayesian inference. This shows us that applying Bayesian inference after the retriever-ranker system indeed helps improve performance.

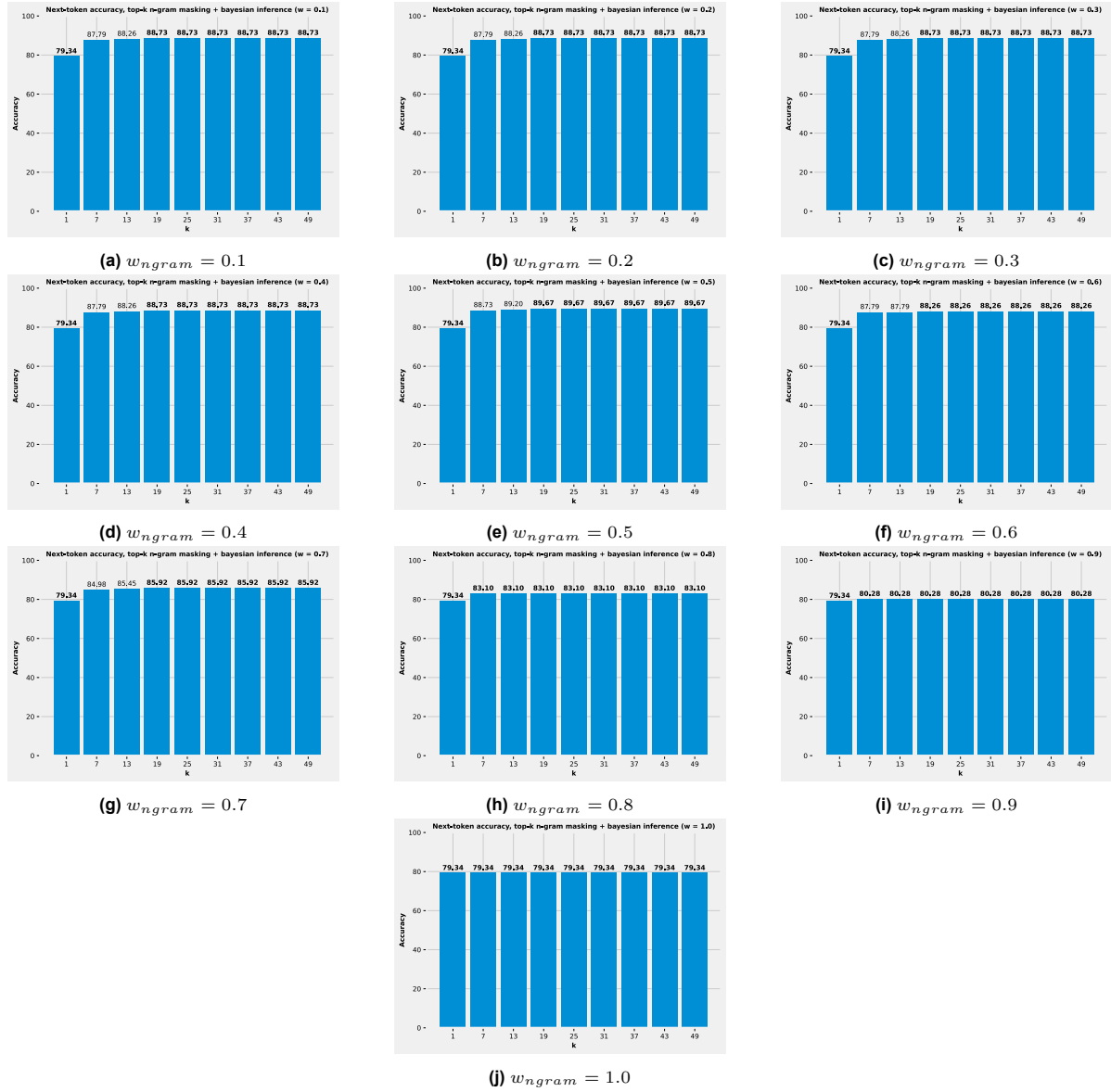


Figure 6.22: Top-k Masking + BI Hybrid Model Next-token Accuracy (Identifiers)

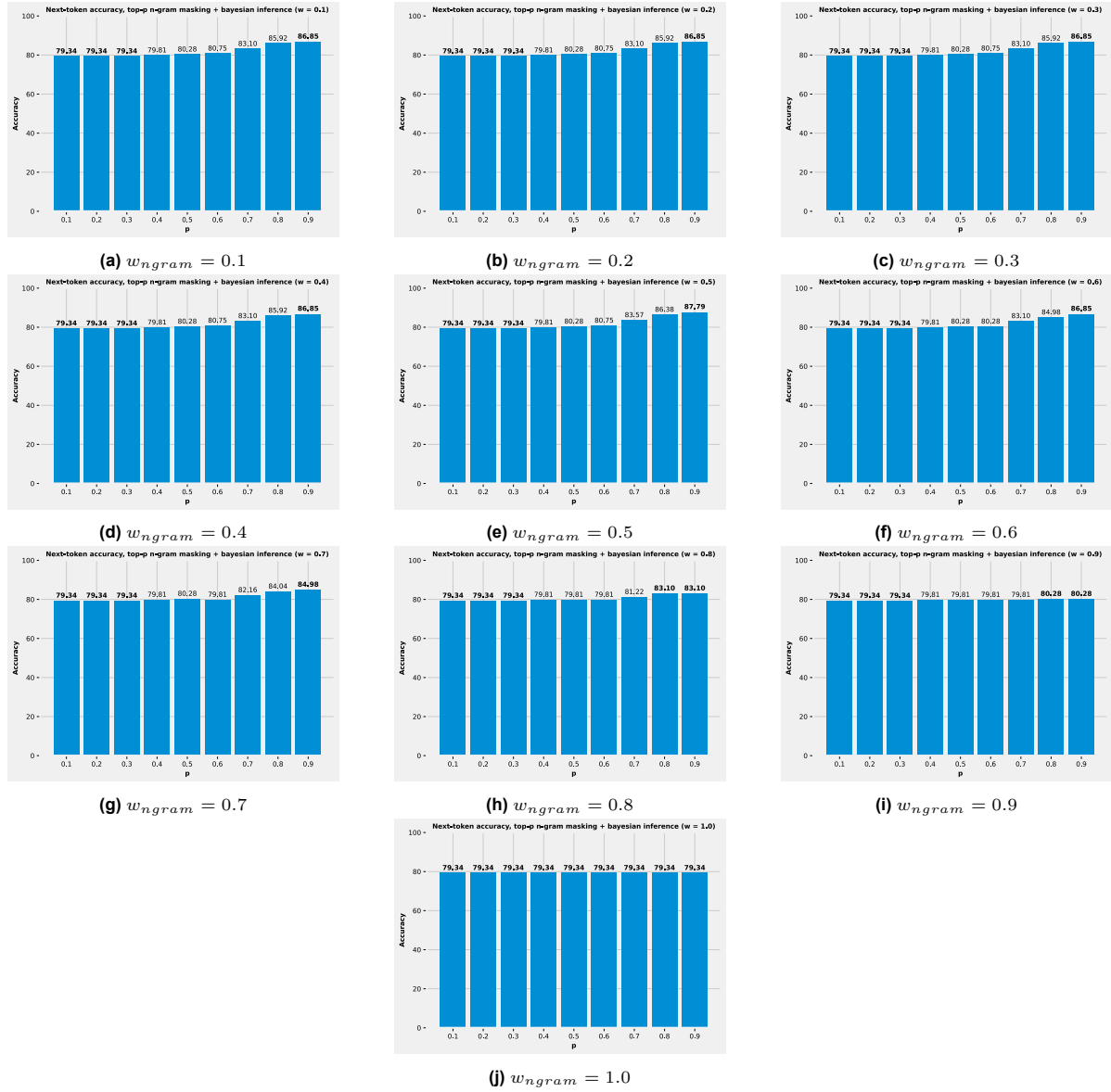


Figure 6.23: Top-p Masking + BI Hybrid Model Next-token Accuracy (Identifiers)

Experiment	Accuracy
None	84.34
Top-1 Threshold-UB	75.90
Top-1 Threshold, $k = 19$	75.90
Top-2D Threshold-UB	75.90
Top-2D Threshold, $k = 19$	75.90
InNgram Threshold-UB	84.34
InNgram Threshold, $k = 19$	77.11

**Table 6.6:** Top-k Masking + Activation Criterion Accuracy Hybrid Model Accuracy (Identifiers)

nevertheless, the performance regression once more shows that top-p masking is not suitable for n-gram retrievers.

**RQ3.3:** Concluding, we find that using top-k masking in conjunction with Bayesian inference can further improve the accuracy of our hybrid model. This finding verifies that project context matters for both retrieval and inference. Once more, top-p masking is shown to be less effective as a result of the probability mass concentration in the n-gram model. The overall improvement in accuracy over the baseline is relatively small (2.41%pt), with a slightly larger improvement in next-token accuracy (5.16%pt). Nevertheless, as our approach has minimal overhead it presents an option for consumers to improve their code completion experience to some extent by improving the context-awareness of code completion models.

### 6.2.6. Conditional Application

The baseline accuracy for identifier prediction is 75.90%. As this is quite high, there are many cases where using a hybrid model is not necessary. While our hybrid model is able to relinquish control to the LLM and does not activate when not predicting identifiers, there is still a considerable gap between the theoretical upper bound on accuracy and the accuracy that was achieved in practice. Being able to predict whether we should use a hybrid model could reduce computational overhead and reduce potential regression caused by the hybrid approach. We investigate three ways to conditionally apply the hybrid model to establish whether it is possible to accurately predict whether a hybrid model should be used. Additionally, we investigate to what extent conditional application is possible without diminishing the accuracy gained through TIC. We explore three distinct ways to conditionally apply a hybrid model using top-k masking.

#### Top-1 Threshold

Our first approach uses the LLM probability distribution to gauge its certainty. If the highest probability  $\max_i(p_i | p_{i-n:i-1})$  is below a threshold  $t$  we deem the LLM to be uncertain, to which we respond by activating the hybrid model. We investigate the consequences this activation criterion has on accuracy and next-token accuracy, and how well this criterion can predict mistakes in the LLM's output.

#### Experiment 6.20: UB on Top-k Masking Accuracy with Top-1 Threshold Activation (Identifiers)

The theoretical upper bound for top-1 threshold is 75.90%, matching the baseline performance. Based on this result, it is clear that using a threshold on the highest LLM probability is not a suitable way to determine whether to activate the hybrid model. We hypothesize that the LLM's lack of project context knowledge makes it overly confident in cases where project context is required; the fact that the LLM only knows about the current file limits its scope, leading to high probabilities due to its limited context.

Figure 6.24 displays the ROC curve for the top-1 threshold, highlighting its ability to predict whether the LLM will produce an incorrect prediction based on different  $t$  values. We observe excellent performance with an AUC of 0.89% and a wide range of attainable TPR-FPR (false positive rate, true positive rate) pairs. This finding shows that being able to predict whether the LLM will be wrong does not imply that the hybrid model can repair potential mistakes. Therefore, to prevent regression, it is favorable to be liberal with the activation of the hybrid model.

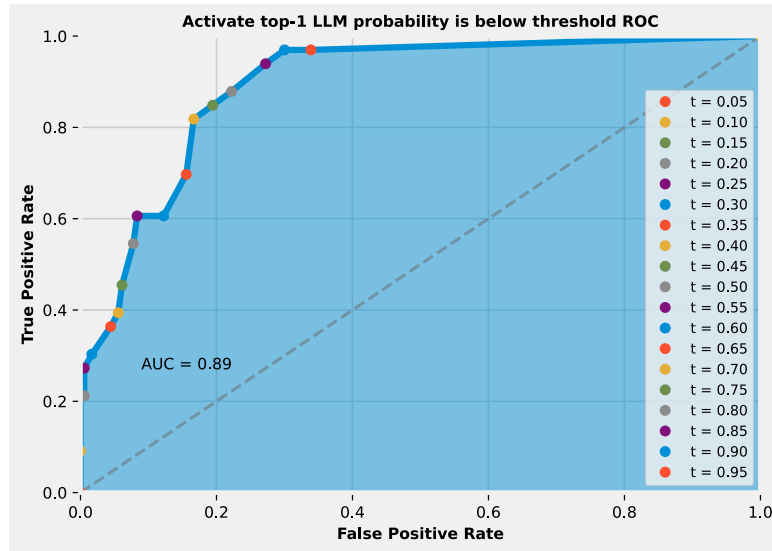


Figure 6.24: Top-1 Threshold Activation Receiver Operating Characteristic (ROC) Curve

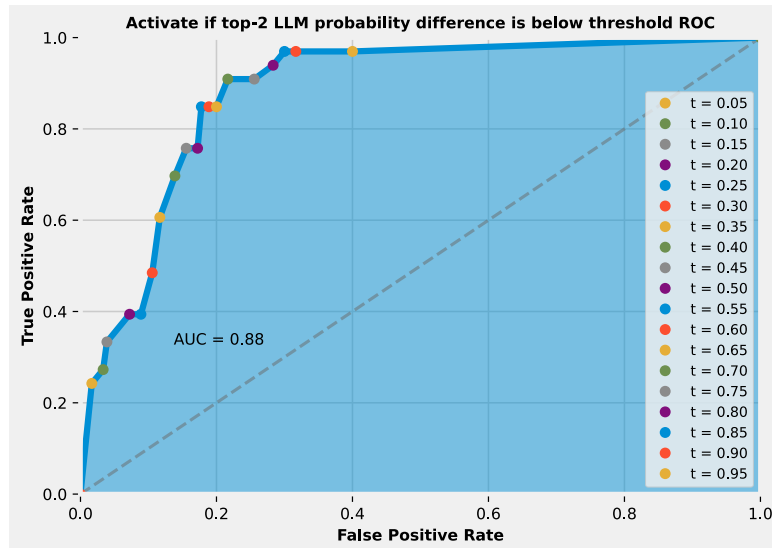


Figure 6.25: Top-2D Threshold Activation Receiver Operating Characteristic (ROC) Curve

### Top-2D Threshold

As an alteration to the top-1 threshold, the top-2D (top-2 difference) threshold considers the difference between the highest two probabilities. If this difference is relatively small, the LLM appears to be uncertain, causing the hybrid model to activate.

#### Experiment 6.21: UB on Top-2D Masking Accuracy with Top-1 Threshold Activation (Identifiers)

We once more establish a theoretical upper bound. As per Table 6.6, the theoretical upper bound for top-2D threshold activation is 75.90%, matching the baseline and top-1 threshold performance. The identical results follow from the fact that the LLM is not able to accurately describe its own certainty in relation to the project context without being able to access the project context, leading to high probabilities.

Similar to the top-1 threshold, the top-2D threshold achieves an excellent ROC-AUC of 0.88 (Figure 6.25), and supports a wide range of TPR-FPR pairs. This confirms our finding that the ability to predict incorrect LLM predictions is not equivalent to being able to predict when the hybrid model should be used.

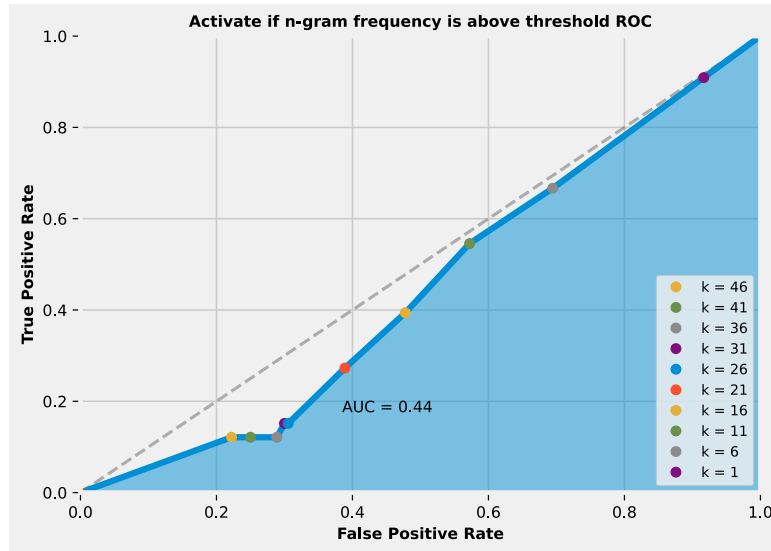


Figure 6.26: In-n-gram Threshold Activation Receiver Operating Characteristic (ROC) Curve

### In-N-gram Threshold

While the previous activation criteria were based on LLM probabilities, the in-n-gram threshold considers frequency of a prefix within the n-gram model. If the frequency of the n-gram input is above some threshold  $t$  the n-gram is likely able to help, thus activates the hybrid model.

**Experiment 6.22: UB on Top-k Masking Accuracy with In-n-gram Threshold Activation (Identifiers)** The upper bound accuracy for the in-n-gram threshold is 84.34%, which matches the upper bound accuracy for the top-k masking method without an activation criterion.

Inspecting the ROC curve (Figure 6.26) shows that the in-n-gram threshold is not effective at predicting whether the LLM will be wrong. In fact, as the AUC-ROC is 0.44 it is worse than random guessing. Despite the low AUC-ROC, the upper bound performance shows that the in-n-gram threshold may still be effective without causing regression. This shows conditionally applying the hybrid model does not require being able to predict whether the LLM will be wrong, likely because the n-gram model is not always able to correct wrong LLM predictions.

**Experiment 6.23: Top-k Masking Accuracy with In-n-gram Threshold Activation with  $k = 19$  (Identifiers)** When  $k = 19$ , the in-n-gram threshold achieves an accuracy of 77.11%. This matches the accuracy of the top-k masking method without an activation criterion, showing that this activation criterion is effective in preserving the performance of the hybrid model.

Despite the success of the in-n-gram threshold, its efficacy at limiting computational overhead of the hybrid model is limited. While using LLM probabilities as activation criterion bypasses the hybrid model altogether, the in-n-gram threshold must query the n-gram model, which is one of the more expensive operations of the hybrid model. Nevertheless, using the frequency of n-gram prefix frequencies as a proxy for n-gram model's ability to predict the next token proves to be a valid approach.

**RQ4:** To conclude, using LLM probabilities as an activation criterion disregards knowledge of project context, leading to overconfidence in the LLM's predictions, subsequently causing under-application of the hybrid model. Conversely, using the n-gram model to determine whether can potentially help the LLM proves to be a more effective approach, albeit with more computational overhead.



# 7

## Run-Time Efficiency

Our implementation of the hybrid global-local approach is designed to be as efficient as possible, i.e., to have minimal impact on the inference time of the code completion model. We perform benchmarks on our test set to determine the overhead of TIC Chapter 6. It is important to understand what effects incorporating a local n-gram model has on the inference time, as higher inference times can decrease productivity even when improving the accuracy of predictions. As an example, retrieval-augmented generation systems that require multiple passes through the LLM to find the best combination of contextual information can be very slow, leading to very high accuracy, but also very high inference times.

We benchmark TIC by running our hybrid model with top-k masking on our test set.<sup>1</sup> We capture timing information for 1) applying softmax on LLM logits, 2) retrieving n-gram train inputs, 3) constructing the n-gram model, 4) querying the n-gram model, and 5) combining the probability distributions of the LLM and n-gram model. We also capture the overall time it takes to determine what proportion of time is overhead rather than LLM inference, and the time it takes to apply the complete the adaptive code completion system (i.e., everything excluding LLM inference).

The results of our benchmarking (Figure 7.1) show that our approach does introduce overhead, but that this overhead is small compared to the time it takes to apply the LLM. In total, our approach takes on average 29.72 ms of the 731.68 ms required to run an iteration of TIC inference (4.06%). The most expensive part of our approach is the initial training of the n-gram model, taking up roughly 31.85 milliseconds. This is an insignificant amount of time, especially considering the fact that this only has to be run once. As for processing time on each round of inference, we see that querying the model is most expensive, followed by adjusting the n-gram model to the current input, followed by combining LLM and n-gram probabilities. Nevertheless, this overhead can nearly be eliminated in its entirety by running it in parallel with LLM inference. Parallelization would only result in the mixing of n-gram probabilities and LLM probabilities being the only form of overhead.

**RQ5:** The low overhead of our approach shows that it is viable for use in real-world scenarios such as in IDE plugins. Additionally, there is enough margin to increase n-gram size or to add additional processing steps.

---

<sup>1</sup>Benchmarks are run on an Apple MacBook Pro with a 16-core M3 Max processor and 64 GB of RAM.

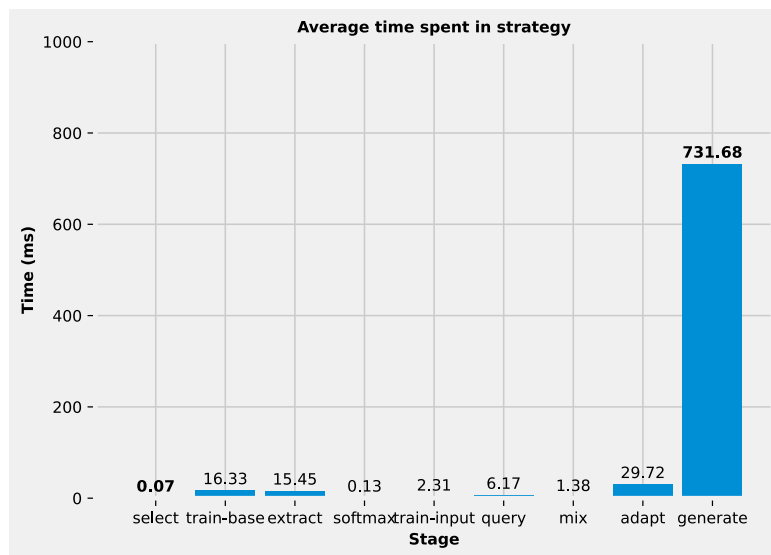
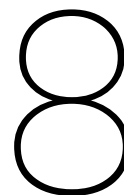


Figure 7.1: Benchmark Results



# Discussion

This section outlines the main findings of this thesis, and discusses the implications of our results and the limitations of our approach.

## 8.1. RQ1: LLM + N-gram Bayesian Inference

The results of our experiments show that the gap between the abilities of LLMs and n-grams is considerable. This indicates that the use of n-gram models may be limited in practice due to the superiority of LLMs.

While our experiments emulate the setup used by Tu, Su, and Devanbu, we did not explore every aspect of their experiments. For instance, we did not explore constraining the size of the n-gram model's training data. The primary reason for this is the small size of the project context available in our dataset. This leaves the question of how much context is required for the n-gram model to be of most use. Namely, too much training data may make the n-gram model too general, while too little training data may make the n-gram model's relevance limited.

## 8.2. RQ2: Parts of Code Dependent on Project Context

Our findings indicate that different parts of code are connected to project context to different degrees. We explored identifiers and function invocations, and showed that n-gram models trained on identifiers can help LLMs be more accurate when predicting identifiers. The same did not hold true for function invocations, as their precise nature does not mix well with n-gram models. The practical implications of these findings are twofold. First, the fact that identifiers can be a useful target for hybrid models shows that more sophisticated hybrid models (e.g., based on language specifications) may be able to improve code completion to a greater degree. Second, the fact that our hybrid approach was not able to leverage function invocations as a target highlights the limitations of n-gram models, once more suggesting that the use of n-gram models may be limited in many practical scenarios.

The primary limitation of our approach is the fact that no parts of code aside from identifiers and function invocations were explored. Interesting future avenues could include exploration of specific types of identifiers. Additionally, partitioning the n-gram model into multiple more specific n-gram models to better target different types of identifiers could prove to be an interesting approach.

## 8.3. RQ3: Targeted Identifier Completion

Our investigation of different hybrid models yielded a number of interesting findings.

### 8.3.1. RQ3.1: Masking Techniques

Our exploration highlighted that masking approaches can be just effective as Bayesian inference, if not more effective. The efficacy of a retriever-ranker system shows the potential for a wide range of alternative hybrid systems. This shows, for instance, that it is possible to integrate a wide range of

retrieval systems into code completion models. Practical examples could include static analysis tools, language servers, or even small fine-tuned language models.

Additionally, our findings verify the limitations of n-gram models due to its vulnerability to OOV issues, as highlighted by the fact that top-p masking proved ineffective.

### 8.3.2. RQ3.2: Masking Components and Project Context

Investigating the impact of using the n-gram model for ranking opposed to retrieving showed that project context is especially important when selecting which tokens may occur in an identifier. This highlights the fact that LLMs that can only access file-context are limited in their ability to help developers write code.

### 8.3.3. RQ3.3: Masking with Bayesian Inference

Experimenting with the application of Bayesian inference after masking shows that the understanding of project context does not only help during retrieval, but also during ranking. This once more highlights the limitations of LLMs when applied in unknown territory.

Additionally, we highlight the fact that the LLM under test was confident in its predictions even though it cannot fully understand the project context. This finding suggests that LLMs could be improved by making them aware of potentially missing context during training. Code LLMs could, for instance, give users an indication when they are missing context. While concrete approaches to achieve this require more research, benchmarks such as the Long Code Arena [43] could be used to adjust the unsupervised learning process: rather than training the model to classify the next token correctly, the model could be trained to distribute probability mass over multiple tokens if project context is required to predict the next token. LLMs trained in this manner could be more effective when combined with hybrid models.

The gap between the theoretical upper bound on accuracy and the accuracy achieved in practice highlights the potential for improvement that can be unlocked by context-awareness. This showcases a weakness of LLMs, and shows limitations of our n-gram-based approach.

While we explored different ways of improving the performance of hybrid global-local models, there are still many variations that could be explored by future research.

## 8.4. RQ4: Conditional Application

Our conditional approach once more highlighted the overconfidence of LLMs in unfamiliar contexts, hinting at the need for additional awareness in LLMs. Additionally, we show that it is possible to apply the hybrid model conditionally without sacrificing accuracy.

Nevertheless, the findings of our conditional approach are limited by the dataset and LLM used in our experiments. Application on other datasets or in tandem with different LLMs could lead to different results.

## 8.5. RQ5: TIC Overhead

Benchmarks of our TIC approach show that the overhead incorporating small n-gram models in code completion is small. Due to the computational requirements of LLMs, training and querying small n-gram models can be achieved without noticeable impacting the latency of code completion.

This finding highlights the lightweight nature of n-gram models, showing that despite their limited power they can be used to bring marginal improvements in practice.

Overall, our experiments showcase the limitations of LLMs in new contexts, while also highlighting the lack of awareness on the part of LLMs. Additionally, we show that retriever-ranker systems form a good basis for hybrid models.

Future work could explore alternative training methods to make LLMs more aware of their limitations, could explore different hybrid models by using more powerful models, or could strengthen our findings by using different LLMs and datasets.

# 9

## Conclusions

In conclusion, this thesis presents a comprehensive evaluation of hybrid LLM-n-gram models that leverage the programming skill of LLMs and the ability of n-gram models to quickly learn project context. The elaborate experiments and analyses result in numerous key insights.

First, we show that the global LLM outperforms the local n-gram model when it comes to capturing localness of code. This highlights the large gap in proficiency between LLMs and n-gram models, as the LLM can only access in-file context whereas n-gram can access all project files. Nevertheless, applying n-gram models to context-sensitive details of code proves to be a viable approach.

We show that targeting identifiers (TIC) can boost code completion accuracy considerably when using an oracle for  $w_{ngram}$ . In practice, a fixed weight leads to an improved identifier completion accuracy of 1.21%pt.

While function invocations provide detailed information that may be relevant to new completions, n-gram models were not able to leverage this information to improve code completion accuracy. The precise nature of function invocations does not mix well with n-gram models, as any deviation from sequences that occur in the n-gram training data verbatim renders the n-gram model ineffective. Additionally, the short context window size of n-gram models increases the difficulty of completing long function invocations.

Moreover, using n-gram models to retrieve relevant tokens and using LLMs to rank these tokens can beat the accuracy of the Bayesian inference approach. This approach shows a potential for further improvements when using an oracle for the number of tokens to retrieve  $k$ , and performs on par with Bayesian inference when using a fixed value for  $k$ . Our experiments indicate that top-k masking is especially effective, while top-p masking is not a viable approach as n-gram distributions are heavily skewed towards the most likely token(s).

Results show that knowledge of project context is especially important in the retrieval phase, as using the LLM to retrieve and n-gram to rank drastically reduces performance. Yet knowledge of project context can also help in the ranking phase. Applying Bayesian inference after the retriever-ranker system pushes accuracy to new heights, achieving an increase in accuracy of 2.41%pt over the baseline.

While conditional application of the hybrid model can save computational overhead without compromising accuracy, our benchmarks indicate that the overhead is minimal in general. This makes our hybrid model suitable for use on consumer hardware in real-world scenarios to provide a small increase in accuracy.

To summarize, we demonstrate that hybrid LLM-n-gram models can improve code completion performance in new environments marginally by leveraging project context understanding when predicting code most sensitive to context like identifiers. We show that n-gram models can be used to improve the context-awareness of code completion models, and highlight the need for LLMs to be more aware of their limitations.

# 10

## Future Work

There are several directions to further explore or strengthen our findings.

### 10.1. Steering with Language Features

Incorporating language features in hybrid models is an interesting approach that seen limited exploration by Agrawal et al. [35]. Future approaches could generalize this idea by integrating directly with the Language Server Protocol (LSP) to use language servers as a knowledge base. This would allow for further enhancements spanning a wide range of languages. Such approach do not necessarily have to be limited to masking approaches, but could also incorporate n-gram models, for instance by constructing many different n-gram models that target specific variable types, or construct names like class names and function names.

### 10.2. Human-in-the-loop

Our hybrid model tries to automatically adapt to environments unfamiliar to the LLM. This system leaves out an important source of information, namely the developer applying code completion. Experimenting with different ways to receive feedback to adjust the model could improve accuracy and could point to problems in our approach that must be addressed to further increase code completion capabilities. A feedback system could for instance be used to automatically change the training data used by the n-gram model, or could change parameter values to correct mistakes.

### 10.3. Evaluating Real-World Scenarios

As our approach was tested on a static dataset it is unclear how well our hybrid model performs in the real world. It is therefore interesting to explore how well this approach performs in practice. This could be combined with research into live parameter tuning; while we were able to find clear optimum values in our different experiments, the optimum may be heavily dependent on the project context. If this is true it is important to be able to adjust these parameters automatically to suit new codebases. This does not necessitate a human-in-the-loop approach; automated A/B testing approach can also prove worthwhile.

### 10.4. Local Light-Weight Multi-Layer Perceptrons

While our local n-grams were able to provide marginal accuracy gains, the architecture itself is rather limiting. As new hardware ships with integrated AI chips more frequently, training small Multi-Layer Perceptrons (MLPs) on consumer hardware becomes feasible. Exploring this approach could lead to performance improvements far beyond the advancements unlocked by n-gram models.

## 10.5. Different Programming Languages

This research used a dataset consisting of Python code. Future work could investigate whether results differ on other programming languages. Additionally, alternative programming languages properties could be explored. Rather than focusing on identifiers, for instance, one could experiment on more constrained categories, such as variable names of specific types, or names of constructs like classes.

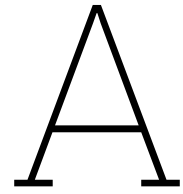
# References

- [1] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. “On the localness of software”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2014, pp. 269–280.
- [2] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).
- [3] Alec Radford et al. “Language Models are Unsupervised Multitask Learners”. In: 2019.
- [4] Tom Brown et al. “Language Models are Few-Shot Learners”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901.
- [5] Mark Chen et al. “Evaluating large language models trained on code”. In: *arXiv preprint arXiv:2107.03374* (2021).
- [6] Raymond Li et al. “StarCoder: may the source be with you!” In: *arXiv preprint arXiv:2305.06161* (2023).
- [7] Loubna Ben Allal et al. “SantaCoder: don’t reach for the stars!” In: *Deep Learning for Code Workshop (DL4C)*. 2023.
- [8] Erik Nijkamp et al. “Codegen: An open large language model for code with multi-turn program synthesis”. In: *arXiv preprint arXiv:2203.13474* (2022).
- [9] Erik Nijkamp et al. “Codegen2: Lessons for training llms on programming and natural languages”. In: *arXiv preprint arXiv:2305.02309* (2023).
- [10] Erik Nijkamp. *Codegen2.5: Small, but mighty*. Oct. 2023. URL: <https://blog.salesforceairesearch.com/codegen25/>.
- [11] Patrick Lewis et al. “Retrieval-augmented generation for knowledge-intensive nlp tasks”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 9459–9474.
- [12] Sept. 2020. URL: <https://ai.meta.com/blog/retrieval-augmented-generation-streamlining-the-creation-of-intelligent-natural-language-processing-models/>.
- [13] Sebastian Riedel et al. *Retrieval augmented generation: Streamlining the creation of Intelligent Natural Language Processing Models*. Sept. 2020. URL: <https://ai.meta.com/blog/retrieval-augmented-generation-streamlining-the-creation-of-intelligent-natural-language-processing-models/>.
- [14] Nelson F Liu et al. “Lost in the middle: How language models use long contexts”. In: *arXiv preprint arXiv:2307.03172* (2023).
- [15] Suchin Gururangan et al. “Don’t stop pretraining: Adapt language models to domains and tasks”. In: *arXiv preprint arXiv:2004.10964* (2020).
- [16] Tim van Dam. *Replication Package*. <https://github.com/AISE-TUdelft/adaptive-code-completion> [Accessed: 2024].
- [17] Philip Gage. “A new algorithm for data compression”. In: *The C Users Journal* 12.2 (1994), pp. 23–38.
- [18] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. URL: <https://aclanthology.org/N19-1423>.
- [19] Colin Raffel et al. “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer”. In: *J. Mach. Learn. Res.* 21.1 (Jan. 2020). ISSN: 1532-4435.



- [20] Tomas Mikolov et al. "Efficient estimation of word representations in vector space". In: *arXiv preprint arXiv:1301.3781* (2013).
- [21] Yinhan Liu et al. "RoBERTa: A Robustly Optimized BERT Pretraining Approach". In: *arXiv e-prints*, arXiv:1907.11692 (July 2019), arXiv:1907.11692. arXiv: 1907.11692 [cs.CL].
- [22] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536.
- [23] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.
- [24] Kyunghyun Cho et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078* (2014).
- [25] Dan Jurafsky and James Martin. "N-gram Language Models". In: *Speech and language processing*. Pearson Education, 2014.
- [26] Jason Wei et al. "Chain-of-thought prompting elicits reasoning in large language models". In: *Advances in neural information processing systems* 35 (2022), pp. 24824–24837.
- [27] Shengyu Zhang et al. "Instruction tuning for large language models: A survey". In: *arXiv preprint arXiv:2308.10792* (2023).
- [28] URL: <https://platform.openai.com/docs/api-reference/chat>.
- [29] Luís Eduardo de Souza Amorim et al. "Principled syntactic code completion using placeholders". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. 2016, pp. 163–175.
- [30] Shuai Lu et al. "Reacc: A retrieval-augmented code completion framework". In: *arXiv preprint arXiv:2203.07722* (2022).
- [31] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. "Repository-level prompt generation for large language models of code". In: *International Conference on Machine Learning*. PMLR. 2023, pp. 31693–31715.
- [32] Huy N Phan et al. "Repohyper: Better context retrieval is all you need for repository-level code completion". In: *arXiv preprint arXiv:2403.06095* (2024).
- [33] Shuyan Zhou et al. "Doccoder: Generating code by retrieving and reading docs". In: *arXiv preprint arXiv:2207.05987* (2022).
- [34] Outlines-Dev. *Outlines-dev/outlines: Structured text generation*. URL: <https://github.com/outlines-dev/outlines>.
- [35] Lakshya A Agrawal et al. "Guiding language models of code with global context using monitors". In: *arXiv preprint arXiv:2306.10763* (2023).
- [36] Shuai Lu et al. "Codexglue: A machine learning benchmark dataset for code understanding and generation". In: *arXiv preprint arXiv:2102.04664* (2021).
- [37] Kelvin Guu et al. "Retrieval augmented language model pre-training". In: *International conference on machine learning*. PMLR. 2020, pp. 3929–3938.
- [38] Md Rizwan Parvez et al. "Retrieval augmented code generation and summarization". In: *arXiv preprint arXiv:2108.11601* (2021).
- [39] Urvashi Khandelwal et al. "Generalization through memorization: Nearest neighbor language models". In: *arXiv preprint arXiv:1911.00172* (2019).
- [40] Ze Tang et al. "Domain Adaptive Code Completion via Language Models and Decoupled Domain Databases". In: *arXiv preprint arXiv:2308.09313* (2023).
- [41] Yangruibo Ding et al. "Cocomic: Code completion by jointly modeling in-file and cross-file context". In: *arXiv preprint arXiv:2212.10007* (2022).
- [42] Junwei Liu et al. "STALL+: Boosting LLM-based Repository-level Code Completion with Static Analysis". In: *arXiv preprint arXiv:2406.10018* (2024).
- [43] Egor Bogomolov et al. "Long Code Arena: a Set of Benchmarks for Long-Context Code Models". In: *arXiv preprint arXiv:2406.11612* (2024).

- [44] Kishore Papineni et al. "Bleu: a Method for Automatic Evaluation of Machine Translation". In: *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, July 2002, pp. 311–318. DOI: 10.3115/1073083.1073135. URL: <https://aclanthology.org/P02-1040>.
- [45] Chin-Yew Lin. "ROUGE: A Package for Automatic Evaluation of Summaries". In: *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, July 2004, pp. 74–81. URL: <https://aclanthology.org/W04-1013>.
- [46] Mikhail Evtikhiev et al. "Out of the bleu: how should we assess quality of the code generation models?" In: *Journal of Systems and Software* 203 (2023), p. 111741.
- [47] Slava Katz. "Estimation of probabilities from sparse data for the language model component of a speech recognizer". In: *IEEE transactions on acoustics, speech, and signal processing* 35.3 (1987), pp. 400–401.
- [48] Irving J Good. "The population frequencies of species and the estimation of population parameters". In: *Biometrika* 40.3-4 (1953), pp. 237–264.
- [49] William A Gale and Geoffrey Sampson. "Good-turing frequency estimation without tears". In: *Journal of quantitative linguistics* 2.3 (1995), pp. 217–237.
- [50] Reinhard Kneser and Hermann Ney. "Improved backing-off for m-gram language modeling". In: *1995 international conference on acoustics, speech, and signal processing*. Vol. 1. IEEE. 1995, pp. 181–184.
- [51] Artem Sokolov. 2015. URL: <https://www.cl.uni-heidelberg.de/courses/ss15/smt/scribe6.pdf>.
- [52] Ismail. "Comparison of modified kneser-ney and witten-bell smoothing techniques in statistical language model of bahasa Indonesia". In: *2014 2nd International Conference on Information and Communication Technology (ICICT)*. IEEE. 2014, pp. 409–412.
- [53] Thorsten Brants et al. "Large language models in machine translation". In: *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*. 2007, pp. 858–867.
- [54] URL: <http://www.ee.columbia.edu/~stanchen/e6884/labs/lab3/x207.html>.
- [55] Gautier Dagan, Gabriele Synnaeve, and Baptiste Rozière. "Getting the most out of your tokenizer for pre-training and domain adaptation". In: *arXiv preprint arXiv:2402.01035* (2024).
- [56] Guidance-Ai. *Guidance-ai/guidance: A guidance language for controlling large language models*. URL: <https://github.com/guidance-ai/guidance>.
- [57] Scott Lundberg. *The art of prompt design: Prompt boundaries and Token Healing*. Dec. 2023. URL: <https://towardsdatascience.com/the-art-of-prompt-design-prompt-boundaries-and-token-healing-3b2448b0be38>.
- [58] Maliheh Izadi et al. "Language models for code completion: A practical evaluation". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 2024, pp. 1–13.
- [59] Ivan Provilkov, Dmitrii Emelianenko, and Elena Voita. "BPE-dropout: Simple and effective sub-word regularization". In: *arXiv preprint arXiv:1910.13267* (2019).
- [60] Frank F Xu et al. "A systematic evaluation of large language models of code". In: *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 2022, pp. 1–10.
- [61] Ziyang Luo et al. "Wizardcoder: Empowering code large language models with evol-instruct". In: *arXiv preprint arXiv:2306.08568* (2023).



# Architecture UML Class Diagram

This appendix highlights the architecture of the code used to construct the hybrid models used throughout the experiments. The code is designed to be modular and open to extension through the use of mixins. Model behavior can be altered by swapping out classes that implement the same methods or by creating new classes that (partially) implement the abstract methods of the `AdaptiverCodeCompletion` class.

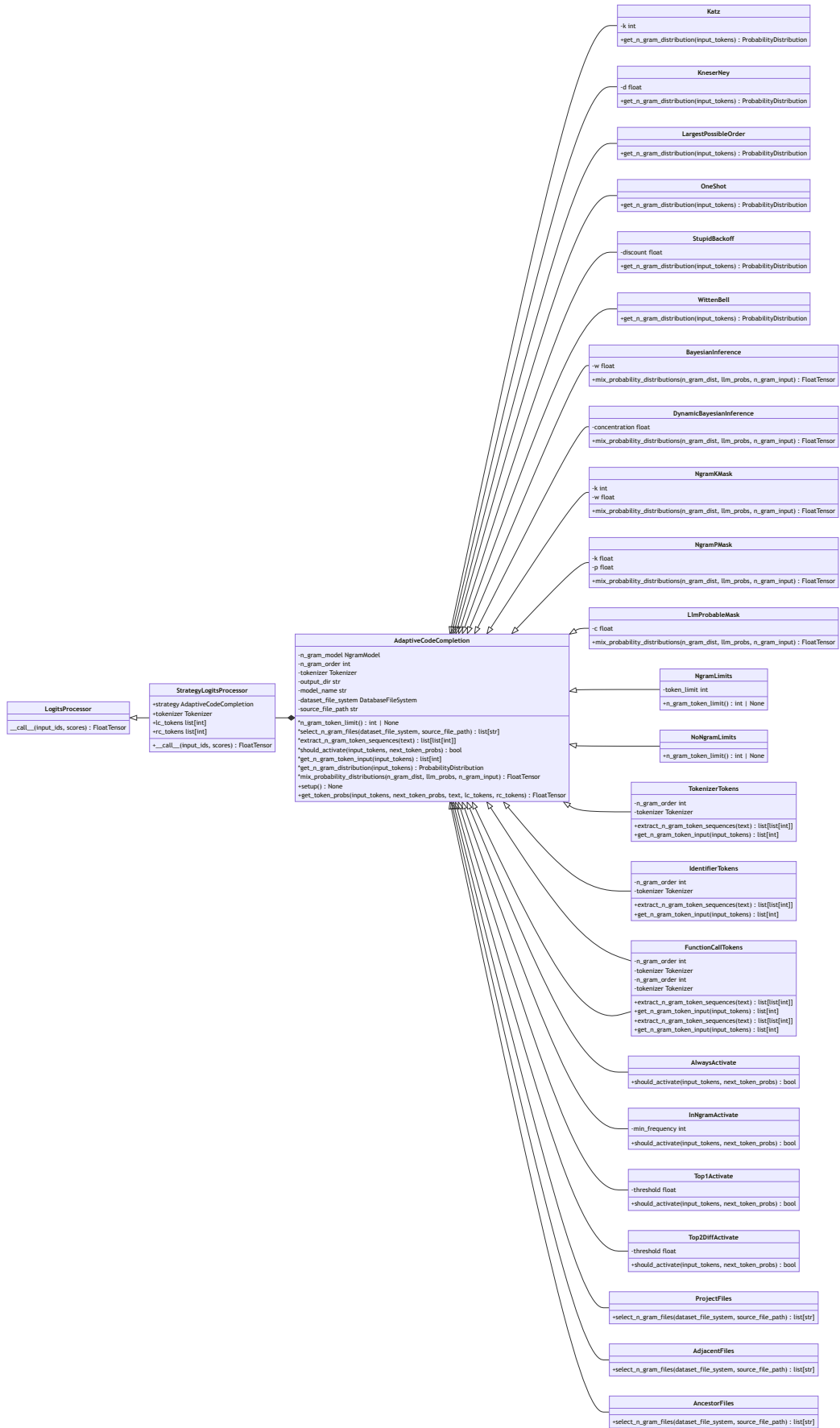


Figure A.1: UML Diagram of the Code Architecture

# B

## Mid-Token Invocation

This appendix includes an experiment that shows the efficiency of a pre-processing and post-processing approach that can be applied to improve performance when code completion models are applied in the middle of a token. This experiment was conducted near the onset of this thesis, and similar approaches have been published over the period in which this thesis was written [55, 56, 57].

In “Language models for code completion: A practical evaluation”, Izadi et al. evaluate the most common causes of code completion failures in real-world settings. A key finding is that out of the 2030 different application-oriented errors, 1173 are due to mid-token invocations. As this study allowed users to invoke code completion in the middle of a token this is not surprising, but can lead to substantially worse code completion performance. Previous works have proposed altering training to address this, e.g., BPE Dropout [59] randomly prevents the merging of tokens to represent cut-off tokens within the training data. However, this approach is problematic as it requires a significantly larger training dataset to support all variations of mid-token invocations. For this reason, we devise a simple approach to alleviate such issues and test its effectiveness on code samples from this paper.

The key issue with mid-token invocations is the fact that the token representation of a sequence will not match the representation models are trained on. During training, models are faced with subsequences cut off at exact token boundaries. In the case of mid-token invocation, the sequence is cut off before the natural boundary of the full sequence. This leads to the model being applied on a token that was not commonly seen in the same context in the training data. To alleviate this issue, we introduce a simple pre-processing and post-processing approach:

First, remove the last  $n$  tokens of the input. Next, run inference, but mask output tokens such that only tokens that match the removed text are allowed. Finally, stop masking once the model has replaced the removed subsequence.

This simple procedure allows the model to choose the token representation it deems most appropriate for the given context, allowing it to change the input to fit the model's training data.

**Experiment B.1: Mid-Token Invocation Pre-Post Processing – Fibonacci** As an example, consider the following input:

```
1 def fibonacci(n):  
2     return n if n < 2 else fibonacci(n - 1) + fibona
```

It is clear that the continuation of this fragment is `cci(n - 2)`, but applying CodeGPT [36, 58] yields `(n - 2)`. The reason for this behavior is clear when investigating the difference in tokenization between `fibonacci` and `fibona`. `fibonacci` is tokenized to `fib on acci`, and `fibona` is tokenized to `fib ona`. It is very likely that `fib ona` is much less common in the training data than `fib on` because models train on complete sequences. Thus, it makes sense to take a step back and allow the model to choose the token representation of `fibona`. Applying the previously described pre-processing and post-processing approach, CodeGPT is able to predict the correct continuation.

Beam Size	Accuracy	Accuracy (Enhanced)
1	1.21	14.55
5	1.98	13.64

**Table B.1:** Mid-Token Invocation Next-Token Accuracy Results (Izadi et al.)

Model	Beam Size	Accuracy	Accuracy (Enhanced)
CodeGPT	1	8.60	9.96
CodeGPT	5	6.13	7.67
WizardCoder-1B	1	7.28	7.44
WizardCoder-1B	5	6.91	6.66

**Table B.2:** Mid-Token Invocation Next-Token Accuracy Results (Unseen Dataset)

**Experiment B.2: Mid-Token Invocation Pre-Post Processing – Izadi et al.** Extending the previous experiment to 450 mid-token invocation samples from Izadi et al. yields the results presented in Table B.1. This clearly shows that performance under mid-token invocation can be drastically improved using the proposed approach. It is also clear, however, that the samples remain challenging for the model.

**Experiment B.3: Mid-Token Invocation Pre-Post Processing – Unseen Dataset** Next, we construct a novel mid-token invocation dataset based on the Unseen Dataset [60]. We employ two different strategies to determine where to invoke code completion:

1. **Random:** Invoke code completion at a random location inside a token.
2. **Unmergable:** Invoke code completion at a location such that the prefix of the token does not occur in any merge rule.

The random strategy emulates behavior more closely matching a real user, while we design the unmergable strategy to be more challenging. We employ a fine-tuned version of CodeGPT [58] and WizardCoder-1B [61]. Our dataset consists of 1 855 samples for CodeGPT and 1 927 samples for WizardCoder-1B. Differences in sample counts are caused by the fact that the two models use different tokenizers.

As displayed in Table B.2, our method does improve performance slightly, but not to the same extent as the previous experiments. We believe this may indicate that our dataset creation strategies are not sufficient to create challenging mid-token invocation samples. Our hypothesis is that the fact that our methods use a tokenizer in the first place may be a factor, i.e., we choose split locations based on a token representation of a full sequence, which leads to a scenario that is easier than real-world scenarios. This highlights that it is not trivial to replicate mid-token invocation issues in a synthetic setting, emphasizing the need for considering before deploying consumer products.