EXPLORING THE DESIGN SPACE OF AI CODE COMPLETION ENGINES

Draft of April 13, 2023 at 23 : 01

BY

PARTH THAKKAR

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2023

Urbana, Illinois

Adviser:

Assistant Professor Tianyin Xu

ABSTRACT

AI based code completion tools such as Github Copilot have recently gained tremendous popularity due to their ability to suggest arbitrary length snippets, improving developer productivity dramatically. However, there is little public understanding of what it takes to build such a tool. In this thesis, we explore the design space of building such a tool. We study the importance of the two key components of such a tool: the Large Language Model (LLM) that predicts the suggestions, and the system around it that feeds it the right context and filters out the bad suggestions. We start by exploring the design of Github Copilot to understand the state of the art, and describe the three key components of Copilot: Prompt Engineering, Model Invocation and Feedback loop. We then study the various factors that affect the quality of the suggestions generated by the LLM. We study both (a) the impact of the context fed to the LLM, and (b) the impact of the LLM itself. For the former, we study the impact including context from other files and code after the cursor along with different methods of context collection and amount of collected context. For the latter, we study the impact of the size of the LLM and the training procedure. Apart from factors affecting the quality of suggestions, we also study the factors affecting the latency of such code completion engines, as low latency is critical for building good code completion engines.

We find that the context fed to the model makes a significant difference in the quality of generated suggestions, and good context collection can improve the quality of suggestions by 11-26% points (20-113% relative improvement) on the exact match metric for one line suggestions. Models that can exploit the context after the cursor can further improve the quality of suggestions by 6-14% points (12-16% relative improvement). Our experiments show that increasing the prompt length beyond a point does not improve suggestion quality significantly, and that 2048-4096 tokens are sufficient. We also find that the size of the LLM has a much smaller impact on the quality of suggestions than other factors such as the context fed to the model and the training procedure used. For example, we found that the SantaCoder model (1.1B parameters) provided better suggestions than the 16B CodeGen-Multi model despite being 16x smaller. This is because the SantaCoder model was trained on a larger dataset and can also exploit the suffix context, unlike the CodeGen model.

Overall, we believe these findings will be useful for the community to understand the design trade-offs involved in building such code completion tools.

ACKNOWLEDGMENTS

Since last several years, I have had a growing desire to be able to work on tools that help programmers. I took a rather elaborate route to get here, working in a few different domains before fully transitioning to this area. I am really happy that I was able to make this transition, and I am very grateful to all the people who have helped me along the way.

First and foremost, I would like to thank my advisor, Prof. Tianyin Xu. He was extremely supportive from the very beginning of my degree, and he has been a constant source of encouragement, guidance and has helped me navigate grad school life at a lot of critical junctures. It is quite funny how we changed the direction of my thesis on a whim after the popularity of my Copilot reverse engineering work. I'm glad Tianyin suggested that. I would also like to thank Prof. Brendan Dolan-Gavitt, whose Fauxpilot project helped inspire a lot of this work.

Apart from people, I am thankful for the existence of some amazing organizations, without which this work would literally not be possible! If not for OpenAI's Codex and Github's Copilot, I would not have been able to do this work. It might seem ridiculous to mention this, but Twitter has been an unbelievably useful way to connect with the vibrant ML community (exactly how I got in touch with Brendan!). Oh, and it would be extremely unfair if I did not mention NCSA for providing an absurd amount of GPUs to run my experiments on!

While this thesis has largely been my focus for the last few months, I also got the opportunity to work on a few other cool projects. I'd like to thank Prof. Reyhaneh Jabbarvand and Prof. Madhusudan Parthasarathy for guiding me on those projects, and Adithya and Razan for being great collaborators!

I would not be here without the unconditional love and support from my mom, dad and Yugma! My time at UIUC would not have been anywhere close to this pleasant if not for Chandy's presence here. I would also like to thank Arun, who's been a great friend since my undergrad days. A big thanks to all my friends at UIUC, Arpitha, Shraddha, Ajay, Karthik, Sachin, for all the fun times.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 MOTIVATION

Recently there has been a dramatic rise of popularity of AI based code completion tools such as Github Copilot [1], Codeium [2], Replit Ghostwriter [3] and Mutable.ai [4] to name a few. Unlike traditional auto-complete tools that can provide only one-word completions [5] in limited situations, this new generation of AI based code completion tools can provide arbitrary snippets as suggestions, by considering the context of the code, including comments written in natural language. While they don't guarantee the correctness of the suggestions, they can be very helpful in speeding up the development process. In fact, according to Github, Copilot writes about 46% of the user's code on average, and for some languages like Java it can be as high as 61%[6]. Despite the popularity of these tools, there is still a lack of understanding of how they work.

These tools are built on top of large language models (LLMs) trained on code, such as OpenAI's Codex [7]. Such models have been trained on terabytes of code data, and have been shown to be able to generate code that can solve some programming problems on benchmarks such as HumanEval [8] and APPS [9]. However, there is more to building a code completion tool than just having a good LLM.

A naive code completion tool can be built by simply prompting the LLM with all the code before the cursor in the current file, and then having the LLM predict the next several tokens in the sequence. But, there is an obvious limitation to this approach – the LLM is not aware of the full context of the code, and only knows about the code before the cursor. As a result, its predictions will often involve 'hallucinated'[10] references to variables and functions that are not defined in the project. Thus, a sophisticated context collection pipeline is needed in order to get meaningful suggestions from the LLM. This context can come not only from the code before the cursor, but also from other files in the project. Furthermore, since majority of the times, programmers are editing existing files, there is also context from the code after the cursor. One could also exploit the cursor's history to identify which parts the user is most likely to be interested in!

However, the amount of context language models can work with is limited. For instance, the 12B parameter Codex model (code-cushman-001) can only work with 2048 tokens at a time. Therefore, it is important to understand how to effectively select relevant parts from the context in a way that is useful for code completion. Even if models improve to deal with larger contexts, we would still want to minimize unnecessary information going into the context, to keep model inference latency low, and also to not confuse the model with irrelevant information. Thus, context collection remains an interesting problem for code completion tools.

Besides context collection, there is also the challenge of serving good suggestions at low latency due to the interactive nature of these tools. While larger language models can in general provide better suggestions, they are also slower to serve predictions for. The latency can be improved by serving these models by utilizing multiple GPUs, which increases the cost of deployment. Thus, there is a three-way trade-off between the quality of the suggestions, the latency of the tool and the cost of the infrastructure. Using smaller models can help improve both the cost and latency, but how it impacts the quality of the suggestions is not well understood.

Prior work has measured these models in terms of how many programming problems they can solve on benchmarks such as HumanEval and APPS. However, the setting of code completion is different from the setting of these benchmarks. First, code completion tools do not have to provide full-function completions to be useful. Users can benefit by having the tool suggest a few words or lines of code at a time. Second, code completion tools have the opportunity to exploit the full context of the code in the project, which can span several files and several thousands of lines of code. Thus, it is not clear how well a model's performance on benchmarks such as HumanEval and APPS translates to the code completion setting.

1.2 OVERVIEW OF RESEARCH QUESTIONS

Given such unknowns, the answer to the question *"What does it take to build an AI-based code completion tool?"* is not clear. We try to answer this question in this thesis. We start by dissecting the architecture of Github Copilot to understand the state of the art in this field. We then explore the design trade-offs involved in building such a tool by asking the following research questions:

- How important is context collection for the task of code completion?
- How do different context collection strategies impact the quality of the suggestions?
- Do longer prompts always lead to better suggestions?
- What is the impact of model size and training procedure on the quality of the suggestions?

- What kind of deployment infrastructure is needed to serve code completion tools at low latency?

To guide our study, we build a new dataset fitting the code completion setting, called the Code Completion Benchmark (CCB). We then use this dataset to answer the above questions. We find that context collection can make a significant difference in the quality of the suggestions. We also find that the quality of the suggestions does get impacted by the model size, but other factors such as the training procedure, the amount of data used for training and the amount of context used for prompting play a more important role. If these factors are fixed, then bigger models do lead to better suggestions – but we only observed marginal improvements in the quality of the suggestions in the models we studied. This suggests that simply scaling up the models may not be the best way to improve the quality of the suggestions. This is consistent with recent results from the machine learning community which has shown that models need to be trained on much more data than the typical amount used so far [11]. Since bigger models are both slower and more expensive to serve, it is more prudent to train smaller models with better training procedures and more data.

We also studied the impact of backend infrastructure on the latency of the tool. We found that the latency of the tool can indeed be improved by using multiple GPUs, but there are diminishing returns due to increased communication cost. We found that the for a fixed model and deployment setting, inference latency is a function of two factors: the size of the prompt, and the desired output length. Since having a sufficiently sized prompt is important for the quality of the suggestions, the main controllable factor is the desired output length. As the latency increases linearly with the output length, it is important to keep the output length small for usability reasons. A natural way to do this is to bias towards providing single-line completions more often. Longer suggestions can be provided in some situations, but a balance between the length, reliability and latency of the suggestions is needed.

1.3 SUMMARY OF FINDINGS

We summarize the key findings of this thesis as follows:

Impact of Context:

- Context from other files in the project can improve the suggestions by 11-26% points in the exact match at first line metric. Depending on the model and other components of the prompt, this can be between 20-113% relative improvement over not using this context.

- The manner of collecting context from neighboring files is important. Using static analysis for collecting context may be an attractive proxy for filtering out irrelevant context, but our results show that using simple text matching across more files is more effective. This is because static analysis is not perfect, and may miss a lot of useful context.

- Models that can handle suffix, such as code-davinci-002 and SantaCoder, can benefit from suffix context by 6-14% points in the exact match at first line metric. This is a 12-60% relative improvement over not using this context, depending on the model and the presence of context from other files. This context appears to be less effective on its own than context from other files.

- Increasing context size of models beyond 4096 tokens does not meaningfully improve the quality of suggestions, at least with current models and prompting strategies. 2048 tokens is a good balance between quality of suggestions and inference cost.

Impact of Model:

- Small models can easily outperform larger models if trained better. Larger models are not necessarily proportionally better than smaller models. For example, the 1.1B SantaCoder model outperforms much larger CodeGen models, and is competitive with Codex models if it is allowed suffix context while Codex models are not.

- Larger models, when trained and prompted identically to smaller models provide better suggestions, but the difference is not necessarily large. We see this in both the CodeGen and the Codex series. For example, the largest CodeGen model (16B) outperforms the smallest CodeGen model (2B) only by 2-4% points depending on the language. Similarly, the largest Codex model (175B) outperforms the smaller 12B Codex model by 2-3% points when both are prompted identically.

- Finetuning models on specific languages negatively affects their performance on other languages. For example, the Codegen-Mono models are obtained from the Codegen-Multi models by finetuning on Python. This finetuning improves Python performance but hurts performance on other languages. As one might expect, the larger models are hurt less than the smaller models by this finetuning. E.g., the 16B Codegen-Mono model is 2% worse than the 16B Codegen-Multi model on Go, while the 2B Codegen-Mono model is 18% worse than the 2B Codegen-Multi model.

Impact of Backend Infrastructure:

- Longer suggestions are slower to generate, and are more prone to errors. Thus, both the latency and the reliability of the suggestions can be improved by biasing towards providing short (e.g., single-line) suggestions more often. However, a balance between the length, reliability and latency of the suggestions is needed.

The rest of the thesis is organized as follows:

- Chapter 2 provides background on AI based code completion tools and related work.
- Chapter 3 describes the architecture of Github Copilot.
- Chapter 4 describes the methodology behind the suggestion quality study.
- Chapter 5 describes the results of the suggestion quality study.
- Chapter 6 studies the impact of backend infrastructure on the latency of the tool.
- Chapter 7 concludes the thesis and discusses future work.

CHAPTER 2: BACKROUND

In this chapter, we first describe Large Language Models, which are the foundation of the new generation of AI based code completion tools. We then describe related work in the area of code completion using LLMs, particularly work that uses LLMs effectively to generate code completions.

2.1 LARGE LANGUAGE MODELS FOR CODE

Large Language Models are a class of machine learning models that are trained on large amounts of text data. They are often able to generate text that is indistinguishable from human written text. These models are trained using a technique called language modeling, which is the task of predicting the next word in a sequence of words. These models tend to be fairly large, having several billions of parameters, as it has been widely believed that scaling up the model size is the key to improving the model's performance. However, increasing the model size has its own challenges, as it requires a lot of compute power both for training and inference.

There has been a long line of work applying machine learning techniques to improve the software development lifecycle. However, the recent explosion in machine learning applications for software development can be attributed to the OpenAI's Codex model[7]. Codex is an LLM trained on a large corpus of publicly available code. Originally, Codex was trained on python code, but it has since been extended to support other languages, and now powers Github Copilot. There are two main variants of Codex: (a) code-cushman-001 (b) code-davinci-002. These models differ not only in their parameter size (12B and 175B parameters respectively), but also in their context window (2048 tokens vs 8096 tokens).

Apart from Codex, many other generative models of code have been trained[12], including many open-source models such as CodeGen models[13] and PolyCoder[14]. The evaluation of most of these models has focused on (a) generating code from natural language comments, and (b) generating full function completions. Popular benchmarks for generative code language models include HumanEval[8] and the APPS dataset[9]. However, these benchmarks do not focus on the task of code completion, but rather on the task of generating full functions/programs from natural language comments. However, as discussed in the previous chapter, code completion is a different task, and requires a different set of capabilities and evaluation metrics.

LLMs that come closer to the task of code completion include InCoder[15], Santacoder[16] and the Davinci variant of Codex. These models can not only generate code given a prefix, but can also take into account a suffix. By design, these models are more tailored to the task of code completion, as most of the time developers are editing existing code, and not writing new code from scratch. Even while writing new code, developers typically do not write code in a top-down manner, allowing models that support suffix to exploit more context.

Such models that can do infilling[15, 17, 18, 19] are not drastically different from the models that only deal with prefix. The only difference is that the training data of infilling models comprises of examples that have both prefix and suffix encoded in the prompt. For example, the sentence “The quick brown fox jumps over the lazy dog” could be encoded as “[PREFIX]The quick[SUFFIX]lazy dog[MIDDLE]brown fox jumps over the”. Given enough such training examples, the model can learn to fill in the middle of text sequences. Furthermore, training on such examples does not cause the model to lose its ability to generate text from left to right.

2.2 PROMPT ENGINEERING FOR CODE COMPLETION

Prompt engineering [20] is the task of formulating a prompt that is suitable for the model to generate useful completions. In particular, the prompt should be able to capture the context of the code completion task, and should be able to convey the intent of the developer to the model. This is a challenging task because of the restriction on the context window of the model. Recent work on code completion using LLMs has recognized the importance of prompt engineering, and has proposed several techniques to improve the quality of the prompts.

Text-similarity based prompting: This approach involves generating the prompt by finding the most similar code snippet(s) in the codebase to the code that the developer is currently editing. The implementations typically generate sliding-window based snippets from the codebase by utilising the recently accessed files or similar heuristics. These snippets are then compared to the code under development by using text based similarity metrics such as jaccard similarity. This approach is used by Github Copilot. Chatper 3 describes this approach in detail. This is a very general approach, and can be used with any model and any language.

Static Analysis based prompting: Some works have proposed to use static analysis to generate prompts. For example, in [21], the authors analyze the codebase to generate a graph of entities, such as files, global variables, classes and functions and the interactions between them. The authors then use this graph to generate prompts. This approach requires some amount of language-specific engineering in order to generate the graph. They reserve a fixed amount of prompt tokens for context from this graph, and the rest of the prompt tokens are used to encode the code under development. Besides static analysis based prompt generation, they also propose a modified context *encoding* scheme that allows encoding much more context than a simple text-based encoding. In particular, they represent each entity in the graph as an embedding vector, and use this vector in the prompt, by passing the embedding layer.

Retrieval augmented prompting: In [22], the authors propose a retrieval based approach to prompt generation. They first slice the codebase into several snippets of fixed size, and generate vector embeddings for each of the snippets. While generating a completion, snippets most similar to the current code are fetched from the database, and are used to generate the prompt. Since these snippets are only similar to the incomplete code, whereas more relevant snippets would be those similar to the actual completion, they perform a second round of retrieval to find snippets that are similar to the proposed completion. This combination of retrieval-based generation and generation-based retrieval helps them generate better completions. However, this can introduce a significant amount of latency because of two rounds of generation.

Learned prompt generation: In [23], the authors propose a learned prompt generation approach. First, they use static analysis to identify various ‘prompt sources’, such as parent class, imported files, siblings, child class, etc. In a way, this is similar to the approach proposed in [21]. From each of these prompt sources, they decide what type of snippet is to be extracted, e.g., all member declarations from a class, method names, method names and bodies, etc. Since there are many such prompt sources, and each prompt source can have multiple snippet types, the authors used a learned model to rank the prompt sources and snippet types. The best ranked prompt source and snippet type are then used to generate the prompt. To train this ranking model, they generate a dataset of large number of prompts, and label them as successful or not based on whether the LLM used to generate the completion was able to generate a correct completion. This approach is language agnostic, and can be used with any LLM.

CHAPTER 3: ANALYZING GITHUB COPILOT

In this chapter, we describe the architecture of Github Copilot, a state of the art code completion tool. In particular, we largely focus on the client side of the system, and describe how it works. In particular, we describe the following aspects of the client: (a) *prompt engineering*, (b) *model invocation*, and (c) *feedback loop.* We obtained these details by reverse engineering the VSCode extension of Github Copilot [24].

3.1 OVERVIEW

There are two main components of Github Copilot:

Client: An editor plugin (e.g., a VSCode extension) that (a) collects whatever a user types, and context from the project, (b) and formulates a "prompt" for LLM in the backend. Whatever the model returns, it then displays in the editor as a suggestion. The client also sends feedback to the backend, which is used to both measure the effectiveness of Copilot, and also to improve the model (if permitted by the user).

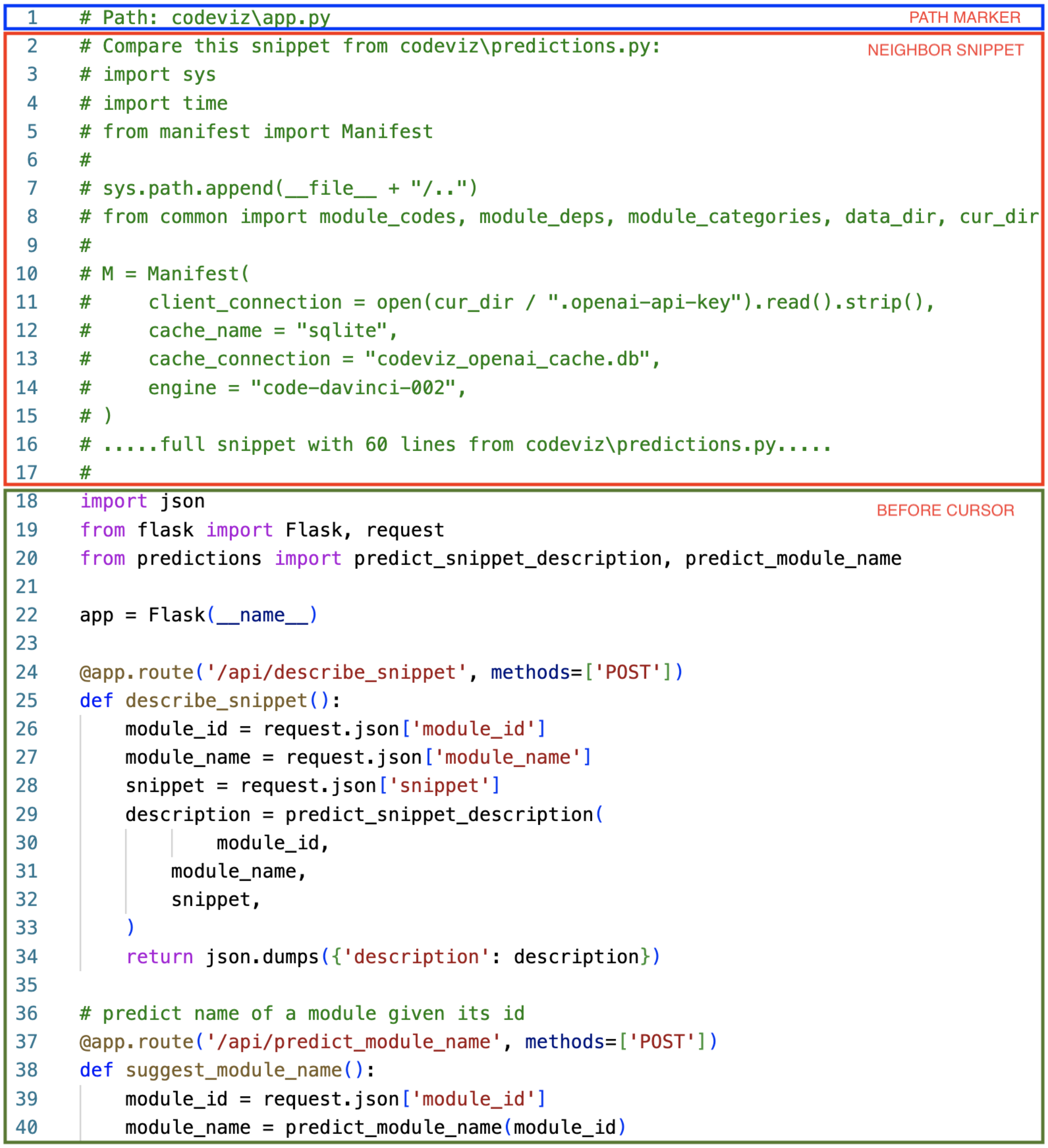
Model: A large language model derived from OpenAI's Codex model. This model is called 'code-cushman-ml' (and has similar variants), suggesting that this is a model derived from the 12B parameter Codex model. However, the difference is that this model also supports infilling, unlike the publicly available 12B parameter Codex model. Note that this model is deployed on the cloud, and is not run locally on the user's machine.

We now focus our attention on the client, and describe how it works.

3.2 PROMPT ENGINEERING

The client is responsible for collecting the context from the project, and formulating a prompt for the model. Having a good prompt is important for the model to generate useful suggestions, as otherwise, the model will be forced to guess variable names and function names that may not be present in the project. On the other hand, the prompt cannot be longer than what the model can handle (2048 tokens), and thus the client must balance what information to include in the prompt.

Components of the prompt: Copilot's prompt consists of two parts: (a) the prefix, and (b) the suffix. The prefix is a combination of information from multiple sources, such as (a) name of the current file, (b) relevant snippets from neighboring files and (c) the code before the cursor. The suffix is the code after the cursor. As prompt size is limited, both prefix and suffix are truncated to fit within 1548 tokens. Copilot retains the remaining 548 tokens for the model to generate suggestions. Figure 3.1 shows an example of the prefix portion of the prompt generated by Copilot.



```
# Path: codeviz\app.py
# Compare this snippet from codeviz\predictions.py:
# import sys
# import time
# from manifest import Manifest
#
# sys.path.append(__file__ + "/..")
# from common import module_codes, module_deps, module_categories, data_dir, cur_dir
#
# M = Manifest(
#     client_connection = open(cur_dir / ".openai-api-key").read().strip(),
#     cache_name = "sqlite",
#     cache_connection = "codeviz_openai_cache.db",
#     engine = "code-davinci-002",
# )
# .....full snippet with 60 lines from codeviz\predictions.py.....
#
import json
from flask import Flask, request
from predictions import predict_snippet_description, predict_module_name

app = Flask(__name__)

@app.route('/api/describe_snippet', methods=['POST'])
def describe_snippet():
    module_id = request.json['module_id']
    module_name = request.json['module_name']
    snippet = request.json['snippet']
    description = predict_snippet_description(
            module_id,
        module_name,
        snippet,
    )
    return json.dumps({'description': description})

# predict name of a module given its id
@app.route('/api/predict_module_name', methods=['POST'])
def suggest_module_name():
    module_id = request.json['module_id']
    module_name = predict_module_name(module_id)
```

Figure 3.1: An example of the prefix portion of a prompt generated by Copilot

Prompt generation pipeline: The prompt generation pipeline used by Copilot involves several steps. First, the current file's path and language are identified, along with the 20 most recently accessed files that are of the same language. A percentage of the prompt tokens to be dedicated to the suffix is decided upfront, typically 15%. For computing the prefix, Copilot identifies a set of "prompt elements" that can be included, such as path and language markers, snippets from recently accessed files, imported files, and the current file. The elements are ranked based on their importance and relevance score, using techniques such as Jaccard similarity to compare snippets' relevance to the current file. The top elements are then selected to fit within the remaining prompt size using a greedy approach. The suffix part of the prompt is set to the code after the cursor and can start from right after the cursor, or from the following function or block. It is truncated to fit within the remaining prompt size. If the tokens dedicated to suffix cannot be fully utilized, then prefix is expanded to fill up the remaining prompt size.

For extracting snippets from recently accessed files, Copilot has various strategies to decide how to extract candidate snippets. The default strategy involves splitting the recently accessed file into sliding windows of fixed sizes, and then computing the Jaccard similarity between the current file and each window. Then, top K windows are selected, and added to the list of prompt elements. Some strategies also involve using indentation to identify blocks of code instead of unconstrained windows. It is unclear to what extent each strategy is used.

3.3 MODEL INVOCATION

There are many interesting aspects of how Copilot invokes the model. While Copilot has two modes of invoking the model, one inline while typing, and the other on-demand. We focus on the inline mode, as it is the most interesting and used aspect of Copilot.

Optimizing for low latency: Copilot requests for 1 suggestion by default, at temperature 0. This is to ensure that the model returns a suggestion quickly. However, if the user requests for more suggestions, Copilot requests for 2-3 suggestions, at temperature 0.2. This is to ensure that the model returns more suggestions, but at the cost of latency. The on-demand mode of Copilot requests for 10 suggestions at a time, thus trading off latency for the number of suggestions.

Single-line vs Multi-line suggestions: Copilot suggests single-line code by default, but when the cursor is at the start of an empty block, it offers multi-line suggestions. While single-line suggestions are quicker and more accurate, Copilot offers multi-line suggestions when the user is likely to insert a block of code, as they are more useful.

Cancelling requests: Copilot debounces the requests to the model, and cancels the previous request if the user types more characters. This is important as model invocation is expensive, and the user may type more characters before the model returns a response.

Caching: Copilot applies aggressive caching of the model responses. This is useful as typing is a very iterative task and the user may move back and forth on the same line. Cached responses help avoid redundant model invocations, while keeping the apparent latency low.

Filtering out poor requests: Copilot has a client-side model that filters out requests that it deems has low chances of resulting in useful suggestions. This model, called "Contextual Filter" is a very small linear regression model based on features such as (a) the current file's language (b) whether the previous suggestion was accepted (c) duration since previous decision (d) last character before the cursor, etc. For example, if the last character before the cursor is a "." (dot) or an opening parenthesis, the model is more likely to allow the request to go through, than if the last character is a ";" (semicolon) or a closing parenthesis. The output of this model is a score between 0 and 1, and Copilot only sends requests to the model if the score is above a threshold (0.15 by default).[24]

Filtering out poor suggestions: Language Models have a common failure mode of generating suggestions that are very repetitive [25]. Copilot applies a simple heuristic to the generated text to detect such cases, and filters them out in the client.

3.4 FEEDBACK LOOP

An important part of Copilot's design is the feedback loop that it has with the user. Copilot collects telemetry data from the user, collecting statistics on the acceptance rate, latency, frequency of cancelling the requests, among many other metrics. This data is used to improve the model. This telemetry framework also helps the Copilot team to perform A/B tests to evaluate the impact of various design decisions. For example, they can help measure the effect of different prefix-computation strategies on metrics like the acceptance rate. This helps them make informed decisions about the design of Copilot.

Measuring suggestion quality: An important part of the feedback look is to meaningfully measure the usefulness of the suggestions. While the simplest metric is the acceptance rate, it is not a good metric for measuring the quality of the suggestions. That is because the user may choose to accept the suggestion and perform a small edit, rather than typing the entire suggestion. For example, if the suggestion is `print('hello')`, the user may choose to accept the suggestion and then change it to `print('hello world')`. This is a common behavior, and the acceptance rate does not capture this.

To measure the quality of the suggestions, Copilot instead periodically measures the edit distance between the accepted suggestion and a window of code around the insertion point. If the word-level edit distance (normalized to be between 0 and 1) is below a threshold (0.5 by default), then the suggestion is considered to be 'still in code'. This measurement happens at exponentially increasing intervals of 15s, 30s, 2min, 5min and 10min. The wide range of intervals is to ensure that the measurement is not too noisy.

Potential data for future training: Furthermore, if the user has not opted-out from sharing their code for product improvement, Copilot also collects a datapoint that could be used for training the model. This datapoint is a pair of two things – a hypothetical prompt that is generated for the inserted position, and the actual code that the user has inserted (regardless of whether the user modified an accepted suggestion, or rejected the suggestion and wrote their own code). This datapoint can be used for training, however, it is not clear what further filtering mechanisms may be applied on this stream of data.

CHAPTER 4: MEASURING COMPLETION QUALITY: METHODOLOGY

In this chapter and the following chapter, we focus on measuring the quality of code completion, regardless of the latency. In this chapter, we first describe the experimental setup and the research questions we are interested in. Next, we describe the dataset we used for our evaluation. In the next chapter, we discuss the results and limitations of our evaluation.

4.1 MOTIVATION

There are many factors that affect the quality of suggestions provided by the code completion engine, but they can be broadly classified into two categories: (a) *model factors* and (b) *prompting strategy factors*. A good model is necessary for good suggestions, but it is not sufficient. For example, the best possible model can still produce bad suggestions if the prompt does not have sufficient information, forcing the model to hallucinate. Similarly, a good prompting strategy can still produce bad suggestions if the model is not good enough.

Model factors include the size of the model and the quality of the training data. The general wisdom is that bigger models are better, but there are many results from the ML community that show that smaller models can beat bigger models. Similarly, the quality of training data is important as well, not just the quantity. Prompting strategy factors revolve around the decisions on what to include in the prompt. The simplest strategy is to include everything before the cursor into the prompt. Clearly this is not the best strategy as there are useful clues in imported files, recently accessed files and content after the cursor. But which of them are important? It's not clear.

It is interesting to note that understanding the effect of prompting strategy factors can help guide model development as well. For instance, if we find that the quality of the suggestions improves by adding certain type of information to the prompt, then we can finetune the model to better utilize that information.

4.2 RESEARCH QUESTIONS

We are interested in answering the following questions about the quality of code completion:

- **RQ1. Importance of different types of context:** Context for code completion can come from different sources, such as current file before the cursor, after the cursor, and even from other files in the project. How much information do these sources provide? How does the quality of suggestions change by including these different types of context?

- **RQ2. Importance of prompt length:** LLMs typically have a limited number of tokens they can keep in memory (context length). Models with longer context length can potentially provide better suggestions due to more available context. However, beyond a point, additional context may not necessarily be helpful, and would only increase the latency and inference cost. Understanding the optimal prompt length can help guide model development.

- **RQ3. Impact of model size**: While the general wisdom is that bigger models are better, is that really the case? Even if so, by how much? If there is not a significant difference, then it may be better to use smaller models to reduce latency and inference cost. And perhaps different training strategies can be used to improve the quality of smaller models.

- **RQ4. Impact of training procedure**: How does the quality of suggestions change with the quality of the training data? For example, the Codegen models from Salesforce are of two varieties - (a) 'multi' and (b) 'mono'. The multi models are trained on 6 different programming languages apart from 15% of Github data, while the mono models obtained by further finetuning the multi models on a large amount of Python data. Thus, the mono models have seen more overall data – does that improve, hurt or not affect the quality of suggestions for other languages? Santacoder from BigCode on the other hand has been trained on just 3 languages, how does it perform when used for other languages?

Note that for RQ1, we are interested not only in how those factors affect the quality of suggestions, but also how much information they provide. For example, if the ground truth completion uses a function defined in another file, then the context from that file may be very important. However, the that function's usage is already part of the prompt, then the context from that file may not be very important. Since prompt collection from other files is not straightforward, it can be useful to know if the context from other files is important or not.

4.3 EXPERIMENTAL SETUP

To evaluate the quality of code completion, we need to simulate how a developer would use the tool. Therefore, we assume the developer is writing code in an editor like VSCode, and has potentially multiple files open that are relevant to the current task. This is not perfect, because not all relevant files may be open, and there may be many irrelevant files present as well. However, this is a reasonable approximation of how a developer would use the tool.

Our experiments consist of various *scenarios*, where each scenario consists of the following:

- MainFile: A The file that the developer is currently editing.
- HiddenRange: A region in that file that the developer has not yet written.
- OpenedFiles: A set of files that the developer has opened in the editor.

Measurement procedure: The code complete tool is expected to help the developer complete the code in the HiddenRange. We simulate this by triggering the code completion tool at the start of the HiddenRange and collecting the suggestion. We then compare the suggestions with the actual code that the developer would have written. We compare the suggestions with the actual code using exact match, levenshtein distance at both keystroke level and word level to measure the usefulness of the suggestions. Note that many completion tools provide single line suggestion as well as multi-line suggestions. Intuitively, longer suggestions have a higher chance of being useful, but they are also more likely to be incorrect. To quantify the usefulness of multi-line suggestions, we measure the score for different lengths of the suggestions, i.e., we measure the score for the first line, the first two lines, the first three lines, etc. up to 10 lines.

Prompt Collection Framework: Since the prompt generation strategy can be arbitrarily complex and can invoke arbitrary editor APIs, we used VSCode's UI-testing framework to (a) automatically launch a VSCode instance with appropriate code completion tool installed as an extension, (b) open the files in the scenario, (c) position the cursor at the start of the HiddenRange, and (d) trigger the code completion tool. We then collected the prompts generated by the tool. These prompts can then be independently evaluated by themselves, as well as be passed to various models to evaluate the quality of the suggestions. This approach is very flexible and can be used to evaluate any code completion tool. Figure 4.1 shows the prompt collection framework.

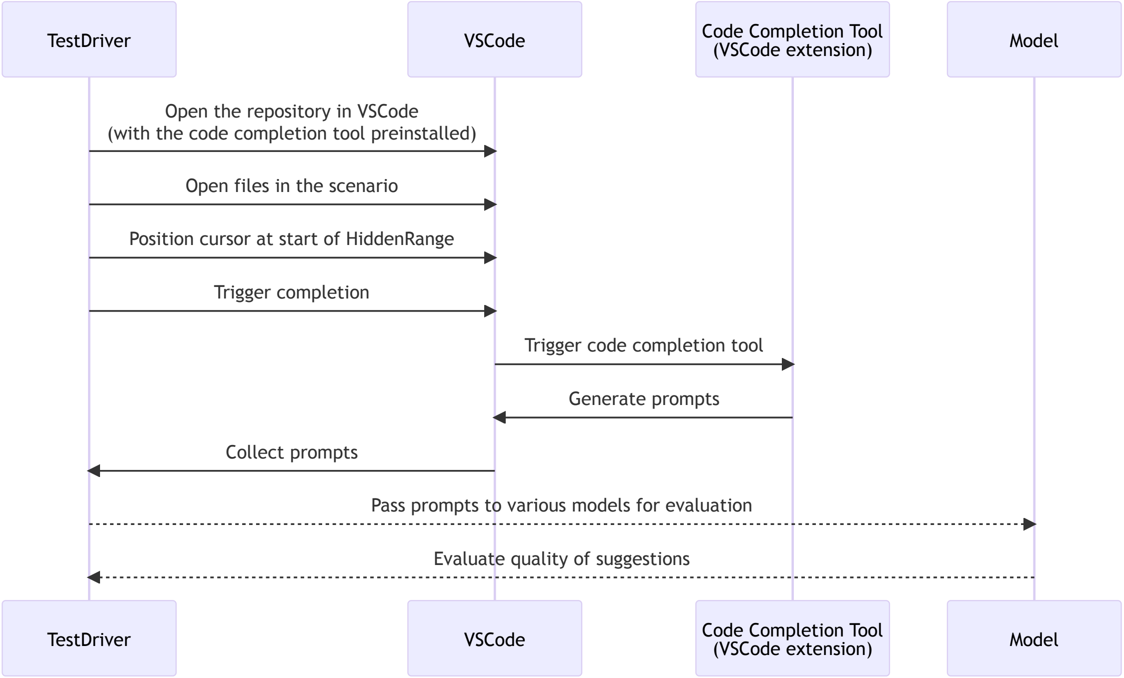

TestDriver
VSCode
Code Completion Tool (VSCode extension)
Model
Open the repository in VSCode (with the code completion tool preinstalled)
Open files in the scenario
Position cursor at start of HiddenRange
Trigger completion
Trigger code completion tool
Generate prompts
Collect prompts
Pass prompts to various models for evaluation
Evaluate quality of suggestions
TestDriver
VSCode
Code Completion Tool (VSCode extension)
Model

Figure 4.1: Prompt Collection Framework

4.4 DATASET

Collecting repositories: For our evaluation, we collected a dataset of 9.5K scenarios from 5 popular programming languages: Java, Python, Go, Rust and Javascript. To collect the dataset, we used the Github Search API to identify repositories between 100 and 200 stars (as a proxy for quality), sorted by the number of commits since 1st January 2023 in order to get the most recently updated repositories. We then cloned the top 50 repositories in this list for each language, totalling to 250 repositories. For each of these 50 repositories, we identified the files that were modified since 1st January 2023 and limited ourselves to extracting scenarios from these files. From each of these files, we extract no more than 5 scenarios to ensure that the dataset is diverse. We also limit the number of scenarios per repository to 60, to further ensure diversity. This results in a total of 9.5K scenarios, with roughly 2K scenarios for each language (with the exception of Javascript, which has 1.6K scenarios).

Data contamination prevention: We emphasize that the reason to collect scenarios from files that have been recently updated is to prevent the dataset from being contaminated by the code that LLMs such as Codex may have memorized [26, 27]. All code models we study in this work have been trained on code from before 2023, and therefore, this dataset is a good proxy for code that the models have not seen before.

Extracting scenarios: To extract a scenario from a file, we randomly mask out a region of code and use the masked region as the HiddenRange. We control the length of the HiddenRange to be between 1 line and 40 lines. Then, using a language server for the respective language, we identify the symbols used in the HiddenRange, identify the files they're defined in, and mark those files as OpenedFiles. We do this because many tools only consider snippets from open files, and opening these files can help the tool to identify relevant snippets from those files. Using a language server here is important as it is a largely reliable way of collecting the symbols and their definitions in a file. Note that the code completion tool is free to ignore these files, and also free to use any other files in the repository.

CHAPTER 5: MEASURING COMPLETION QUALITY: RESULTS

In this chapter we answer the research questions posed in Section 4.2 by studying the impact of different prompting strategies, prompt lengths and models on the quality of the suggestions. Before we dive into the results, we first discuss the subjects of our study.

5.1 SUBJECTS

There are two types of subjects in our study: (a) Prompting strategies (b) Models.

Prompting strategies: For studying different prompting strategies, we use the prompt collection framework described in Section 4.3 to collect the prompts generated by Copilot. Since a prompt can contain various components, such as context from other files, suffix, etc., we modify Copilot's prompt generation logic to control the inclusion of these components. This allows us to measure the impact of each of these components on the quality of the suggestions. We then pass the prompts generated by Copilot to the models under study to measure the quality of the suggestions. Table 5.1 lists the different prompting strategies we use in our study.

We do not modify any other aspect of Copilot's prompt generation strategy, such as the number of candidates, the token budget for suffix when enabled (15%), etc.

Models: We evaluate the quality of the suggestions over a variety of models with different model sizes, abilities and training datasets in order to understand the impact of these factors on the quality of the suggestions. Table 5.2 lists the models we use in our study. We note that while the training datasets for the models are different, most of them are trained on all languages in our scenarios, with the exception of Santacoder which has not been trained on Go and Rust. Furthermore, the models are all trained on code not later than June 2022, reducing the likelihood of them having memorized code snippets from our test scenarios.

It is important to note that the models we use in our study are not the same as the models used by Copilot in production. We use these models because they are publicly available and have their model characteristics documented. We must emphasize that the suggestion quality results on Copilot's *prompts* are not indicative of the suggestion quality of the end-to-end Copilot system in production.

Now we discuss the results of our evaluation. Note that all our results are focusing on scenarios that have more than 5 characters in the first line of the HiddenRange. This is because the primary metric we use is similarity at the first line, and therefore considering scenarios that are too short would not be meaningful.

Strategy	**Description**
Prefix	The prompt is the content before the cursor.
Prefix+NbrDef	The prompt is the content before the cursor, and additional context, but only from the neighboring files that define symbols used in the HiddenRange.
Prefix+NbrAll	The prompt is the content before the cursor, and additional context, but from all files in the repository.
Prefix+Suffix	The prompt is the content before the cursor, and the suffix of the HiddenRange. Suffix occupies 15% tokens of the prompt.
Prefix+NbrDef+Suffix	The prompt is the content before the cursor, the suffix of the HiddenRange, and additional context, but only from the neighboring files that define symbols used in the HiddenRange.
Prefix+NbrAll+Suffix	The prompt is the content before the cursor, the suffix of the HiddenRange, and additional context, but from all files in the repository.
Long prompts	We vary the length of the prompt. The actual context size used by Copilot is 2048 tokens (1548 tokens for the prompt, and rest 500 for the response). We experiment with context sizes of 1024, 2048, 4096, 6144 and 8192 tokens (reserving 500 tokens for response).

Table 5.1: Prompt generation strategies used in our experiments

5.2 IMPACT OF CONTEXT SOURCES

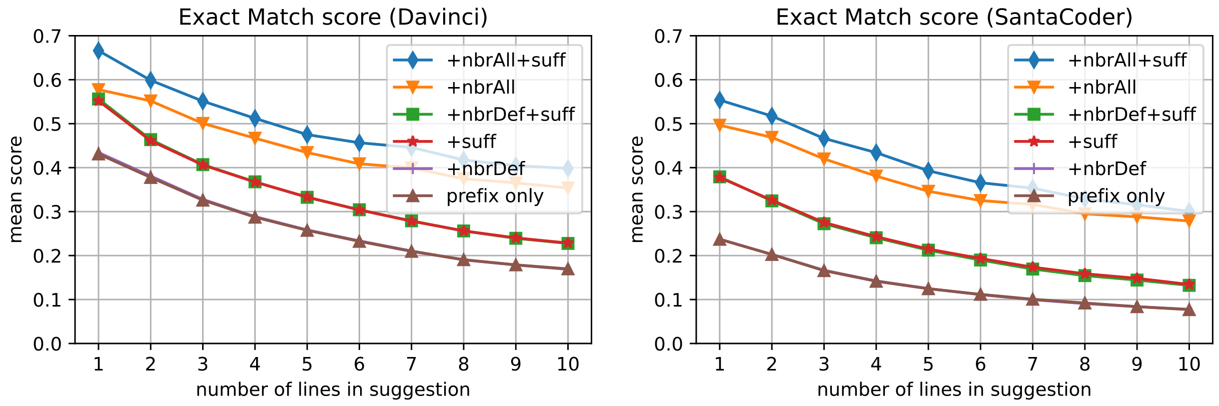
In this section, we evaluate the impact of context from neighbouring files and from below the cursor in the current file. Figure 5.1 shows the impact of these two sources on the quality of the suggestions for the Santacoder and Codex Davinci models. Amongst the model we study, only these models support handling suffix in the prompt.

5.2.1 **Impact of suffix**

The suffix context has a significant impact on the quality of the suggestions. For the Davinci model, adding suffix information to the prompt improves the exact match score of the first line of suggestions from 0.43 to 0.55 (28% relative improvement) in absence of context from other files. For the santacoder model, the improvement is even more dramatic – the same score goes from 0.23 to 0.37 (60% relative improvement!). These trends hold for other metrics as well – mean exact match score over multiple lines of suggestion, character level edit similarity and word level edit similarity. In the presence of context from other files (NbrAll variant), the impact of suffix is reduced, but still significant. Davinci improves from

Name	**Model Size (#Params)**	**Training Data**	**Context Length**	**Handles Suffix?**
santacoder	1.1B	Python, Java, JavaScript	2048 tokens	Yes
codegen-multi	2.7B 6.1B 16.1B	C/C++, Go, Java, JavaScript, Python	2048 tokens	No
codegen-mono	2.7B 6.1B 16.1B	Same as Multi but trained on Python for longer	2048 tokens	No
code-cushman-001	12B	*	2048 tokens	No
code-davinci-002	175B	*	8192 tokens	Yes

Table 5.2: Models used in our experiments



(a) code-davinci-002 (all scenarios)

(b) santacoder (all scenarios)

Figure 5.1: Impact of context sources on the quality of suggestions.

0.57 to 0.66 (16% relative improvement) and santacoder improves from 0.49 to 0.55 (12% relative improvement).

5.2.2 Impact of neighboring files

We explore two main variants when collecting context from neighboring files. In the first variant, *NbrDef*, we exploit the OpenedFiles information from the dataset, to only look at files that define symbols in the HiddenRange. For example, if the HiddenRange invokes a function 'getTweets()', then we look for context in the file that defines this function. The second variant, *NbrAll* looks for context in all the files in the repository. We note that the second variant also dedicates a budget of prompt tokens for this context (we use 1000 tokens by default). We try these two variants to see if focusing on definitions of used functions is

	Exact match @ first line	
model	Prefix only	Prefix + NbrDef
Salesforce/codegen-16B-mono	0.34	0.36 (5.2%↑)
Salesforce/codegen-16B-multi	0.35	0.37 (5.4%↑)
Salesforce/codegen-6B-mono	0.30	0.32 (6.0%↑)
Salesforce/codegen-6B-multi	0.32	0.35 (6.7%↑)
Salesforce/codegen-2B-mono	0.29	0.30 (4.5%↑)
Salesforce/codegen-2B-multi	0.31	0.33 (5.4%↑)
bigcode/santacoder	0.26	0.27 (2.3%↑)
code-cushman-001	0.43	0.46 (7.4%↑)
code-davinci-002	0.46	0.48 (4.5%↑)

Table 5.3: Impact of including context from neighbouring files when NbrDef does pull context from other files. Suffix is not included in the context.

sufficient to provide useful context. Both these variants slice candidate files into snippets of fixed size and use Jaccard similarity between the current file and the snippets to select the most relevant snippets.

The NbrDef variant: The first variant has a negligible-to-modest impact on the quality of the suggestions. Part of the reason for this apparent lack of effectiveness is because only about 43% of scenarios have one or more opened files . When there are zero opened files, there is no context from neighboring files because the Copilot client relies on opened files for extracting the context from other files. Furthermore, even if the neighboring files are present, Copilot may not always decide to use them. For example, if content before the cursor already cannot fit in the prompt budget, then snippets from other files are excluded.

To understand the impact of snippets from neighboring files when the snippets *are* included in the prompt, we zoom into such scenarios. Despite doing so, we find the impact of neighboring files to be rather modest. Table 5.3 shows that the models do benefit from inclusion of neighboring files in the prompt, but the improvement is only 1-3% points in the exact match at first line score. Depending upon the model, this corresponds to 2-7% relative improvement over including only the context before the cursor. Note that in absence of suffix, Santacoder often outputs commented code, despite being correct otherwise. As a result, we addressed this common issue by removing the comments to get a more accurate measure of the quality of the suggestions.

The NbrAll variant: The second variant has a dramatic impact on the quality of the suggestions. For the Davinci model, the exact match score of the first line of suggestions goes from 0.43 to 0.57 (33% relative improvement) in absence of suffix. For the santacoder model, the same score goes from 0.23 to 0.49 (113% relative improvement!). Similarly, in presence of suffix, Davinci improves from 0.55 to 0.66 (20% relative improvement) and santacoder improves from 0.37 to 0.55 (49% relative improvement). It is worth noting that longer suggestions benefit significantly from the inclusion of neighboring files, as can be seen from the slopes of the lines in Figure 5.1.

The difference between these two variants shows that only including the context from the files that define the symbols in the HiddenRange is not sufficient to provide useful context, despite the attractiveness of using static analysis to filter out apparently irrelevant files.

5.2.3 **Amount of neighbor context:**

Since including context from other files has a significant impact on the quality of the suggestions, we explore the impact of the amount of such context included in the prompt. We use the NbrAll variant and vary the number of tokens dedicated to such context. Figure 5.2 shows the impact of the amount of context from neighboring files on the quality of the suggestions. We observe that the quality of the suggestions improves with the amount of context from neighboring files, but since the prompt size is limited, dedicating more context tokens to neighboring files reduces the amount of context from the current file. This trade-off is reflected in the quality of the suggestions. However, dedicating upto 1000 tokens to neighboring files does not negatively impact the quality of the suggestions. It is worth noting that the impact of changing this is relatively small (2 and 4 points in EM score for Davinci and santacoder respectively). This suggests that even a single useful snippet from a neighboring file can be dramatically improve the quality of the suggestions over not having any context.

5.3 IMPACT OF PROMPT LENGTH

We now analyze the impact of prompt length on the quality of the suggestions. For this experiment, we use modified versions of the Copilot client to collect prompts of lengths 524, 1548, 3596, 5644 and 7692 tokens. These numbers correspond to context windows of sizes 1024, 2048, 4096, 6144 and 8192 tokens, where 500 tokens are set aside for the response. Since the only model that can support context windows more than 2048 is code-davinci-002, we limit our experiments to this model. Furthermore, we focus on scenarios from GoLang, Java and Python for this experiment.

Table 5.4 shows the results of this experiment. We observe that the quality of the suggestions improves as the prompt length increases till 6144 tokens, beyond which there is

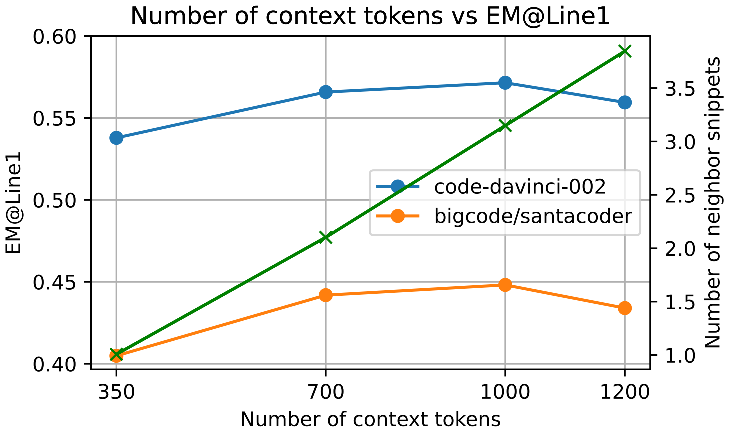


Figure 5.2: Impact of the amount of context from neighboring files on the quality of the suggestions.

no improvement in the suggestion quality. Note that the table includes scenarios where the having larger token budgets *did* actually result in the full budget being utilized. That is, scenarios where the client was allowed to generate prompts of length 8192-500 tokens, but the client generated the same prompt as when the budget was 6144-500 tokens – such scenarios were removed to allow focusing on purely the impact of prompt length on the suggestion quality. This happens when the client is unable to find more context to add to the prompt. If we include such cases the trend remains the same (but we stop seeing meaingful improvements beyond 4096 tokens). This clearly shows diminishing returns from increasing the prompt length.

We performed this experiment with both the NbrDef and NbrAll variants of the prompt. The diminishing returns from increasing the prompt length is observed in both cases, despite the fact that the NbrAll variants packed many more snippets in the prompt. For example, the 8192 context length prompts had, on an average 18 snippets from neighboring files, while the 2048 context length prompts had, on an average, 3 snippets from neighboring files. These results show that at least with current models and prompting techniques, there is no benefit in increasing the prompt length beyond 4096 tokens, and there is only a marginal benefit (4-10%) in increasing the prompt length beyond 2048 tokens which may not be worth the higher latency.

5.4 ANALYZING PROMPTS

To understand why the context from neighboring files has such a modest impact, especially in the presence of suffix, we inspect the prompts generated by the different variants. We use

	Prefix+NbrDef+Suff				Prefix+NbrAll+Suff			
	First Line		Full Suggestion		First Line		Full Suggestion	
Context length	EM	ES	EM	ES	EM	ES	EM	ES
1024	0.50	0.71	0.25	0.50	0.54	0.73	0.27	0.49
2048	0.57	0.76	0.32	0.55	0.66	0.81	0.39	0.59
4096	0.61	0.78	0.34	0.55	0.69	0.83	0.43	0.62
6144	0.62	0.78	0.36	0.57	0.70	0.84	0.43	0.63
8192	0.62	0.78	0.35	0.55	0.70	0.84	0.43	0.63

Table 5.4: Impact of prompt length on the quality of the suggestions. EM = Exact Match, ES = Edit Similarity.

the overlap of identifiers between the prompt and the scenario's HiddenRange as a proxy for the measure of information content of the prompt. For example, if the code in the HiddenRange invokes a function 'get_counts_at_line()', and if the prompt does not contain this identifier, then the model will have a tough time getting this function call right. Of course, if the prompt contains similar identifiers, or conveys the convention of using snake case along with the functionality of the function, then the model may be able to get the function call right. As a result, this is not a perfect proxy for the information content of the prompt, but a good first approximation.

Table 5.5 shows the overlap ratios of identifiers between the prompt and the HiddenRange for the different prompting variants. The overlap ratio is computed as the number of identifiers in the prompt that are also present in the HiddenRange, divided by the total number of identifiers in the HiddenRange. For example, if the HiddenRange contains 10 identifiers, and if the prompt contains 8 of them, then the overlap ratio is 0.8. We compute this ratio for the prefix, suffix and the overall prompt (prefix + suffix). As can be seen in the table, the overlap ratio is the lowest for the prefix-only variant. As additional context sources are added (neighboring files and suffix), the ratio consistently increases. Furthermore, the overlap ratio is consistently higher for NbrAll variants when context from neighboring files is allowed. These observations help explain the better performance of the NbrAll variants. Similarly, the overlap ratio consistently increases by increasing the prompt length, however it quickly saturates after 4096 tokens. This again helps partially explain the saturating performance of longer prompts.

	NbrDef variant			NbrAll variant		
Strategy	Prefix	Suffix	Overall	Prefix	Suffix	Overall
Prefix only	0.81	0.00	0.81	0.81	0.00	0.81
+Nbr	0.83	0.00	0.83	0.87	0.00	0.87
+Suffix	0.79	0.53	0.87	0.79	0.53	0.87
+Nbr+Suffix	0.81	0.52	0.88	0.86	0.48	0.90

(a) Overlap ratios for different context sources

	NbrDef variant			NbrAll variant		
Prompt Length	Prefix	Suffix	Overall	Prefix	Suffix	Overall
1024 tokens	0.72	0.37	0.78	0.71	0.35	0.76
2048 tokens	0.81	0.52	0.88	0.86	0.48	0.90
4096 tokens	0.85	0.60	0.92	0.90	0.55	0.94
6144 tokens	0.85	0.62	0.92	0.90	0.58	0.95
8192 tokens	0.85	0.63	0.93	0.92	0.60	0.96

(b) Overlap ratios for different prompt lengths

Table 5.5: Overlap ratios of identifiers between the prompt and the HiddenRange. Data shown for Python, other languages show similar trends.

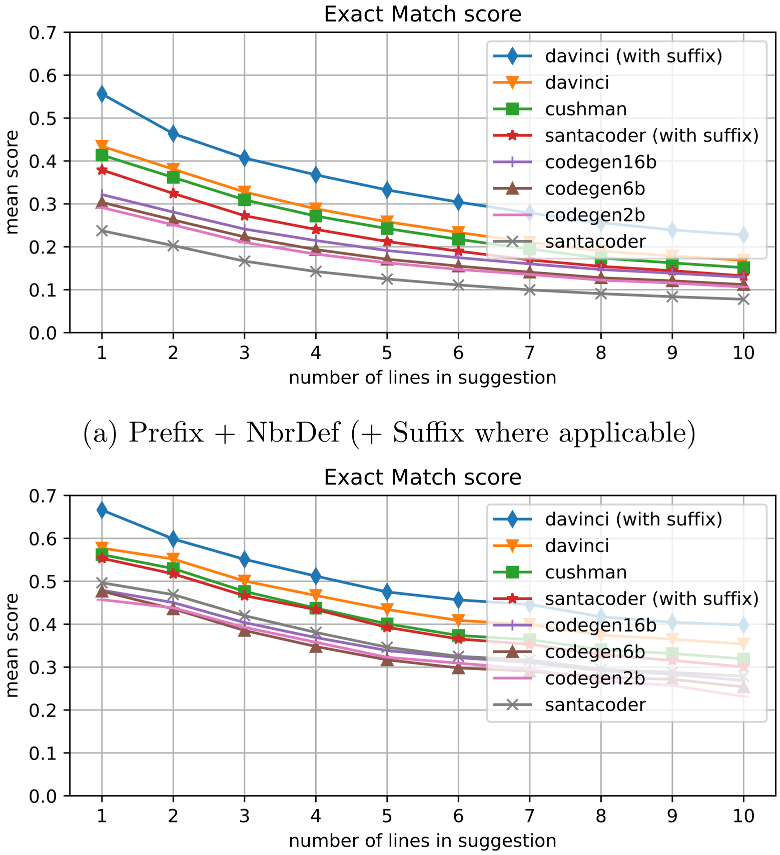
5.5 IMPACT OF MODEL SIZE

We now analyze the impact of model size on the quality of the suggestions. For this experiment, we use the standard 2048 context window and use the best possible prompting strategy for each model. In practical terms, this means, we include the suffix portion when the model has the ability to exploit it (code-davinci-002 and santacoder). For other models, we only include the code before the cursor and context from other files to utilize the token budget optimally.

Figure 5.3 shows the results of this experiment. We observe that the quality of the suggestions improves as the model size increases. This is expected, since larger models have more capacity to learn from the data. However, the impact of model size alone on the quality of suggestions is not very dramatic. For instance, notice the difference between the quality of suggestions from the codegen (multi) models. Despite increasing the size by 8x from 2B param model to a 16B param model, the quality of suggestions only improves by 4% points in exact match at first line score (13% relative). We see a similar trend for the codex models. The Davinci model (175B params) is 14× larger than Cushman (12B params) but shows only a 2% points improvement in exact match at first line score (4% relative) in absence of suffix.

On the other hand, factors other than the model size have a much more significant impact. For example, by using the suffix portion of the prompt, davinci models can gain 9-12% point improvement in exact match at first line score (16-28% relative). Similarly, the santacoder model can gain 6-14% point improvement (12-60% relative) in exact match at first line score. In fact, the santacoder model, a 1.1B param model trained only on Java, Javascript and Python can beat the 16B param Codegen-multi model which has been trained significantly on more languages, when it is allowed to use suffix. Furthermore, all models benefit dramatically when they are allowed to use neighboring files as context using the NbrAll variant. Interestingly, using the NbrAll variant reduces the gap between all the models significantly.

This shows that the model size alone is not sufficient to explain the quality of the suggestions. Other factors such as the training data, the prompt generation strategy, the prompt length, etc. play a very significant role in the quality of the suggestions.



(a) Prefix + NbrDef (+ Suffix where applicable)

(b) Prefix + NbrAll (+ Suffix where applicable)

Figure 5.3: Impact of model size on the quality of the suggestions.

Language Model	Go	Java	Javascript	Python	Rust
Salesforce/codegen-16B-multi	0.42	0.48	0.29	0.35	0.28
Salesforce/codegen-16B-mono	0.41	0.41	0.25	0.37	0.27
Salesforce/codegen-6B-multi	0.40	0.45	0.27	0.33	0.27
Salesforce/codegen-6B-mono	0.32	0.38	0.24	0.40	0.24
Salesforce/codegen-2B-multi	0.38	0.43	0.28	0.33	0.24
Salesforce/codegen-2B-mono	0.31	0.36	0.24	0.37	0.24

Table 5.6: Codegen Multi vs Mono models (Prefix + NbrDef + Suffix)

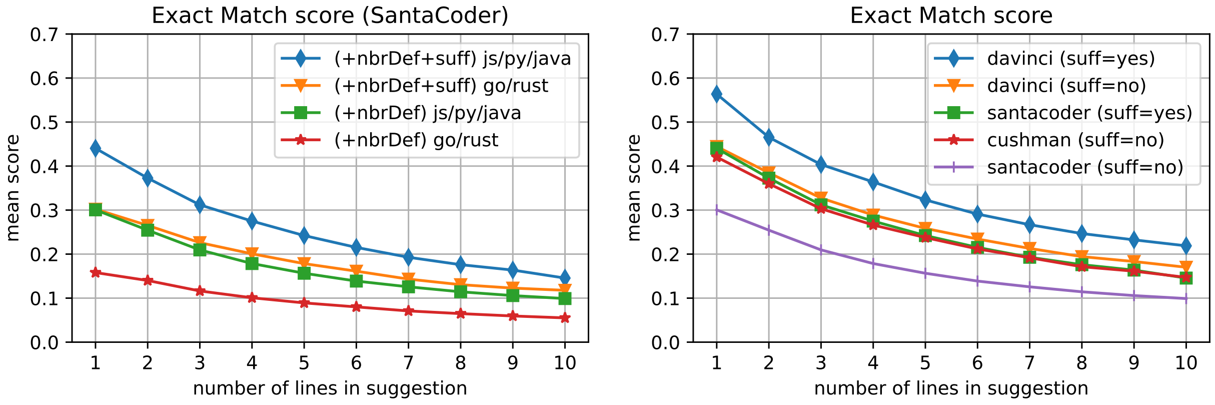
5.6 IMPACT OF TRAINING PROCEDURE

In this section, we discuss two interesting findings related to the model's training procedure. The codegen model series has two variants - (a) 'multi' and (b) 'mono'. The mono variants have been trained on top of the 'multi' models by feeding the models a large amount of Python data. As a result, the mono models have seen significantly more data. However, all the recent updates to their weights have been based on Python data. Thus, there is a possibility that the mono models have reduced performance on other languages. We test this hypothesis by comparing the mono and multi models on the same data. Table 5.6 shows the results of this experiment. We observe that while the mono models have improved performance in Python, there is a significant drop in performance on the other languages. In fact, the 2B multi model is stronger than the 6B mono version with the exception of Python, and in some cases like Java and Javascript, it is better than the 16B mono model as well. This shows that finetuning a model on a specific kind of data towards the end of the training procedure can have the model 'forget' some of the knowledge it has learned earlier. This point should be kept in mind when training large models.

Another interesting finding is that the Santacoder model shows a dramatic improvement in GoLang and Rust scenarios when the suffix is used (Figure 5.4a). This is surprising because the model has not been explicitly trained on these languages. On inspecting some data using BigCode's Dataset Search[1] tool, we found that there are Python, Javascript and Java files that contain GoLang and Rust code snippets, often commented out, or via string literals in code generator functions. This suggests that the model indeed has seen some Go and Rust code, but not enough to learn the language proficiently. In presence of suffix, however, there is a additional context available for the model to exploit, and sees similar performance boost as it does on the other languages.

[1] https://huggingface.co/spaces/bigcode/santacoder-search

If we look at SantaCoder's performance on only JS, Java and Python scenarios (Figure 5.4b), it shows performance comparable to the no-suffix variants of both Codex models. These findings show that the SantaCoder model is a surprisingly powerful model, and scaling it up to more parameters would result in a very strong model.



(a) SantaCoder's performance by language (b) SantaCoder vs Codex on JS/Java/Python

Figure 5.4: Controlling for SantaCoder's training data (Prefix + NbrDef + Suffix)

5.7 TAKEAWAYS

Here is a summary of the main takeaways from our study of quality of suggestions from large language models:

- The quality of prompts has a significant impact on the quality of suggestions. Prompts having context from neighboring files are dramatically more effective than prompts without such context.

- The manner of collecting context from neighboring files is important. Using static analysis for collecting context may be an attractive proxy for filtering out irrelevant context, but our results show that using simple text matching across more files is more effective. This is because static analysis is not perfect, and may miss a lot of useful context.

- Models with the ability to exploit suffix context have a significant edge over models without this ability.

- Increasing context size of models beyond 4096 tokens does not meaningfully improve the quality of suggestions, at least with current models and prompting strategies. 2048 tokens is a good balance between quality of suggestions and inference cost.

- Small models can easily outperform larger models if trained better. Very large models are not necessarily proportionally better than smaller models. For example, the cushman and davinci models perform nearly equally well when fed with the same prompt. Similarly, the 1.1B SantaCoder model outperforms larger CodeGen models, and is competitive with Codex models if it is allowed suffix context, and Codex models are not.
- Finetuning models on specific languages negatively affects their performance on other languages.

CHAPTER 6: INFRASTRUCTURE STUDY

In this chapter, we ask the question, what kind of backend infrastructure is necessary to have a fast and useful AI assistant? We start by discussing the factors that affect latency, and then we present a study of the infrastructure requirements for the code generation models we have been using. We conclude by discussing the implications of our findings.

6.1 FACTORS AFFECTING LATENCY

Since autocomplete systems are used interactively, the latency of the system is a critical factor in the user experience. Human reaction time is about 200 milliseconds, so any latency above this threshold will be noticeable to the user. In some situations, users may be willing to tolerate higher latency in exchange for better suggestions (e.g., longer suggestions, or better quality suggestions). However, in general, the latency of the system should be as low as possible.

The latency of an AI based autocomplete system is largely dominated by the latency of model inference. Factors such as prompt generation and network latency have negligible impact on latency, assuming standard connectivity. We discuss the factors that affect latency in more detail below. All the experiments in this section are performed on a single machine with 8 A10 GPUs.

6.1.1 Inference Backend Choices

The inference backend is the software that runs the model. There are several options for inference backends, including PyTorch, ONNX, TensorRT to name a few. There are various tradeoffs involved in choosing an inference backend, however for the purposes of efficient inference, the most important factors are the latency and memory footprint of the backend. To demonstrate the impact of the inference backend, we compare the latency of the same model running on PyTorch and Nvidia's Triton Inference Server coupled with the FasterTransformer backend [28]. The FasterTransformer backend is a highly optimized inference backend for Transformers models.

Figure 6.1 compares the latency of a 2B and a 6B codegen model running on PyTorch and the FasterTransformer backend, for different prompt lengths and output lengths. As can be seen, the FasterTransformer backend is 1.2-3.5× faster than the Pytorch backend, depending on the model size, prompt length and output length. It should be noted that even for the

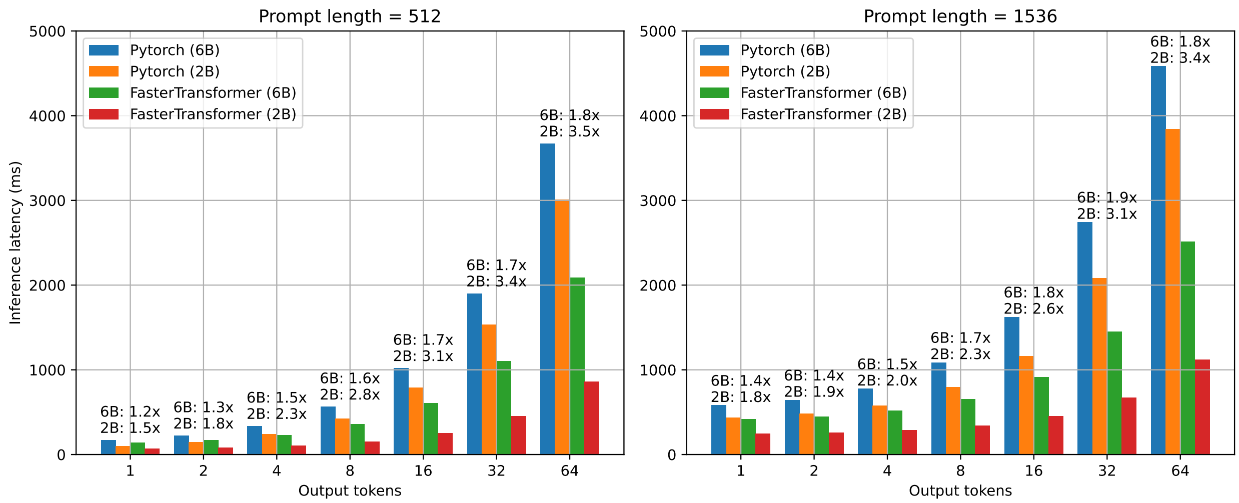


Figure 6.1: Latency comparison of PyTorch and FasterTransformer backends (NumGPUs=1, BatchSize=1)

optimized FasterTransformer backend, the latency can be quite high for big inputs. Note that these are the latencies for a single GPU, serving a single request at a time. In practice, the latency can be improved by using multiple GPUs. We discuss this in more detail in a later section.

6.1.2 Effect Of Sequence Length On Latency

As Figure 6.1 showed, the latency of the model is highly dependent on the length of the input and output sequences. In this section, we discuss the effect of sequence length on latency in more detail.

The inference of a transformer based model involves two stages: (a) context ingestion (b) auto-regressive sampling. The first stage involves processing the prompt and caching the intermediate results, which are repeatedly used in the second stage. In the auto-regressive sampling stage, the model generates one token at a time by conditioning on the previous tokens, and cannot be trivially parallelized. The latency of the auto-regressive sampling stage is therefore proportional to the desired output length. The latency of the context ingestion stage is proportional to the length of the prompt, and is a fixed cost that is incurred for every request, regardless of the desired output length.

Figure 6.2 shows the breakdown of the latency of a 2B and a 6B codegen model for different prompt lengths. The per-token latency in the autoregressive stage shows a mild increase with the prompt length, but the latency in the context ingestion stage increases linearly with the prompt length. Since having a long prompt (i.e., 1500 tokens or more) is

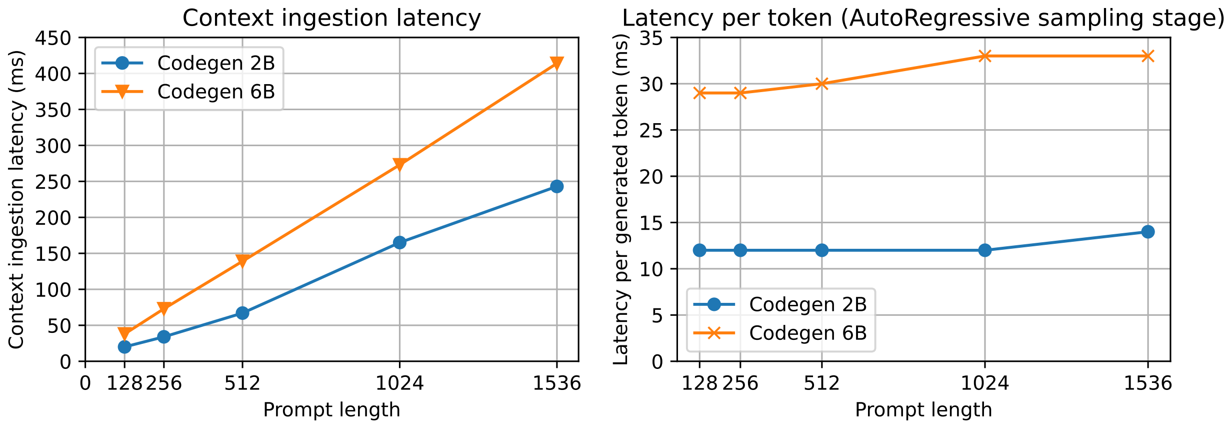


Figure 6.2: Components of inference latency

important for the quality of the suggestions, the main controllable factor that affects latency is the output length. Since longer outputs take longer to generate, the latency of the system can be reduced by reducing the output length. In practice, this suggests a bias towards one-line suggestions over multi-line suggestions. On some occasions, the user may be willing to tolerate a longer latency in exchange for longer suggestions, and balancing this would be an interesting problem to explore in future work.

6.1.3 Sharding

To provide low latency inference for models with billions of parameters, we need to shard the model across multiple GPUs. For efficient inference, model weights are typically sharded horizontally, i.e., each layer of the model is split across multiple GPUs. This form of parallelism, called Tensor Parallelism, allows large matrix multiplications to happen in parallel across devices, at the cost of some communication between the devices. Figure 6.3 shows the latency of a 2B and a 6B codegen model with different number of shards. Evidently, adding more GPUs reduces the latency of the system. However, the latency does not improve linearly with the number of GPUs, because the communication overhead increases with the number of shards. This overhead is more pronounced for smaller models. E.g., for the 2B model, the latency for producing 64 tokens reduces from 504ms to 440ms when going from 4GPUs to 8GPUs (a 13% reduction). However, for the 6B model, the latency reduces from 928ms to 713ms (a 23% reduction). Note that even though the infrastructure cost doubles when going from 4GPUs to 8GPUs while the latency improvement is only 13-23%, this improvement may be a worthwhile tradeoff for some applications, because the impact of a small reduction in latency may still improve user experience significantly.

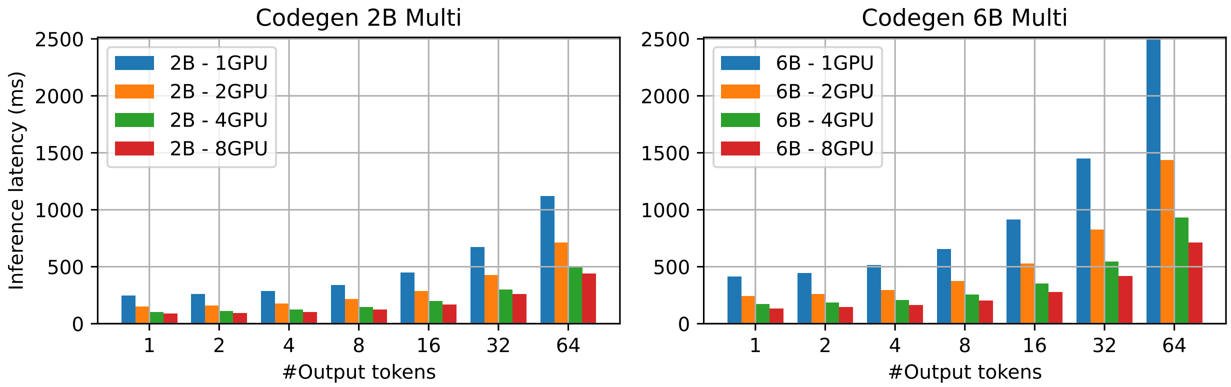


Figure 6.3: Effect of sharding on latency

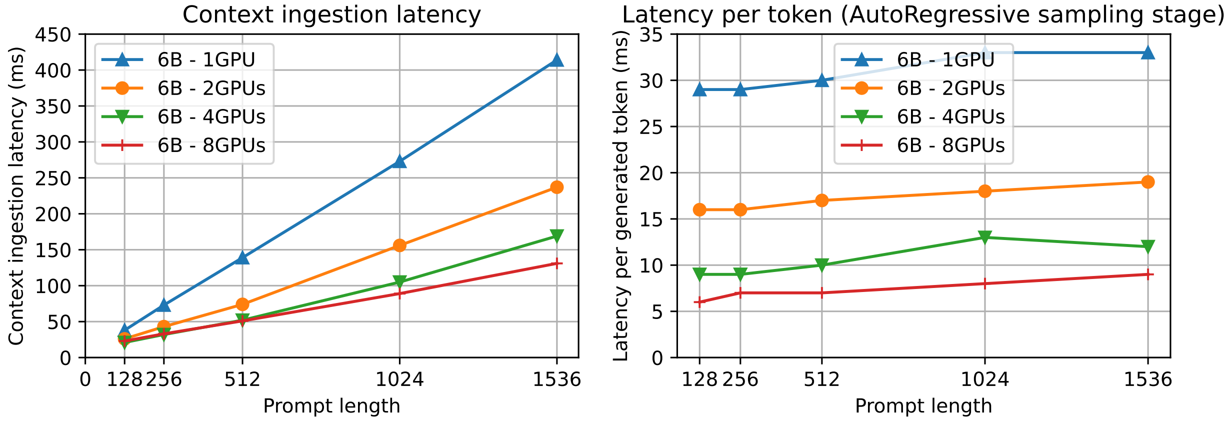


Figure 6.4: Effect of sharding on latency components

Figure 6.4 shows the impact of sharding on the two stages of inference for the 6B Codegen model. Sharding does not help much for short prompts in the context ingestion stage, but helps significantly for long prompts. For the autoregressive sampling stage, sharding helps significantly for all prompt lengths. As the overall latency of this stage is proportional to the desired output length, sharding helps more as the output length increases. For example, with 8GPUs, the latency for producing a token goes from 29ms (with 1 GPU) to 6ms. If 64 tokens are produced, the latency goes from 1856ms to 384ms, which is quite significant.

6.2 ESTIMATING INFRASTRUCTURE REQUIREMENTS

Having looked at the important factors that affect the latency of the system, we now estimate the infrastructure requirements for an AI based code completion system. In the earlier section, we observed that the latency of the system is highly dependent on both the prompt length and the output length. Since the prompt length is required to be long to ensure high quality suggestions, the main controllable factor that affects the latency is the output length. We use this observation to estimate the infrastructure requirements for a code completion system.

We use the dataset of prompts collected by running the Copilot client on the set of scenarios. Every prompt generated by Copilot comes with a list of corresponding 'stop tokens', which are tokens that cause the autoregressive generation to stop. For example, if the stop token is a newline, the model will stop generating tokens once it generates a newline. We use the stop tokens, along with the expected completion code, in order to estimate the output length of the model. We observed that only three sets of stop tokens were used: (a) newline, (b) three newlines together and (c) {\ndef, \nclass, \nif, \n#}. Table 6.1 shows the distribution of the number of tokens in the output for different completion trigger positions. We use these numbers to estimate the infrastructure requirements for a code completion system.

	Completion Triggered	
Percentile	Not at Start of Function	At Start of Function
Median (50%)	10 tokens	19 tokens
75%	22 tokens	74 tokens
90%	88 tokens	164 tokens
99%	250 tokens	250 tokens

Table 6.1: Number of tokens in output for different completion trigger positions

Using the above table, if we want to serve 50% of the responses in less than 200ms and a 2B model is sufficient, then we can use 4 GPUs and meet these requirements. However, a 6B model would need 8 GPUs to generate 10 tokens in around 200ms. Note that these estimates are for running the models on A10 GPUs with FasterTransformer backend. With better GPUs such as A100 variants, more efficient inference backends, aggressive quantization techniques [29, 30, 31] and many other techniques for efficient inference [32] we can reduce the infrastructure required to meet the latency requirements.

CHAPTER 7: CONCLUSION AND FUTURE WORK

In this thesis, we explored the design space of building an AI based code completion tool. We covered the design of Github Copilot, and studied the design trade-offs involved in building such a tool. We found that the quality of the suggestions generated by the Large Language Model (LLM) can be significantly improved by feeding it the right context. At the same time, we found that feeding a larger amount of context to the LLM stops helping after a point, and that context sizes in range 2048 to 4096 tokens may be sufficient. We also found that the size of the LLM has a much smaller impact on the quality of suggestions than other factors such as the context fed to the model and the training procedure used. This is encouraging for the community, as it means that we may not need extremely high amount of resources to build a strong code completion tool.

We also studied the factors affecting the latency of such code completion engines, discussed the effect of various factors such as the inference backend, model size, context length, desired output length, parallelism etc. Along with this, we also analyzed the prompts generated by Copilot for over 9500 tests, and found that for an overwhelming majority of the cases, the desired output length was short. This ensures that the latency for majority of the suggestions will be quite low because for a fixed prompt length, the overall latency is linearly related to the desired output length. This is an interesting detail that can be used to improve the latency of code completion tools. However, at times users may desire longer suggestions, for example at the start of a function, or similar code block. In such cases, the latency may be high, but the user may be willing to wait longer as well. This is an interesting design trade-off that can be explored in future work.

To perform this study, we built the first large scale dataset for code completion spanning 5 programming languages and over 9500 tests, along with an evaluation framework to evaluate the quality of the suggestions generated by any real code completion tool. We built this dataset because the existing benchmarks for LLMs for Code, such as HumanEval and APPS, are not suitable for evaluating code completion tools. They focus on evaluating single-function or whole-program generations based on natural language prompts. However, the setting of code completion is quite different, as the prompts are not natural language, but incomplete program files with several other files in the project. The desired output is also different – single line completions and multi-line completions are both acceptable. We believe that this dataset and the evaluation framework will be useful for the community to improve the quality of code completion tools.

Beyond Code Completion: There is tremendous opportunity for utilizing Large Language Models in improving not just code completion, but all aspects of software development The most exciting of all is the ability to interactively work with any software artifact. This could be interactively writing a complex piece of code, or an interactive walkthrough of a complicated codebase, or even an interactive debugger. There are already some efforts underway to realize this vision, such as Github's Copilot Chat[33] and Cursor[34]. These tools provide a chat based interface in the editor. However, there are several challenges in building such tools. Context collection is again a key challenge, as the model's context size is limited, whereas the codebase can be very large. Similarly, fast inference is also critical for such tools, however the users may be willing to wait longer for the suggestions due to the difference in the setting. Apart from these challenges that carry over from code completion, there are also new challenges. For instance, reliability becomes a key challenge in such tools because they provide longer suggestions, which are time consuming to verify. Code completion, on the other hand, mostly provides short one-line or single-function suggestions, which can be easy to verify. Another interesting challenge is finding the right way of integrating such tools into the existing development workflow. For example, should the LLM be able to install new packages and run arbitrary commands on the system? Allowing these can unlock new ways of developing software, but given the low reliability of LLMs and their vulnerability to adversarial attacks (such as the prompt injection attack [35]), this can pose serious security risks. We believe that these challenges are exciting opportunities for the community to explore.

REFERENCES

[1] Github, “Github copilot,” 2023. [Online]. Available: https://github.com/features/copilot/

[2] Codeium, “Codeium,” 2023. [Online]. Available: https://www.codeium.com/

[3] Ghostwriter, “Ghostwriter,” 2023. [Online]. Available: https://replit.com/site/ghostwriter

[4] Mutable, “Mutable,” 2023. [Online]. Available: https://mutable.ai/

[5] Microsoft, “Intellisense,” 2023. [Online]. Available: https://learn.microsoft.com/en-us/visualstudio/ide/using-intellisense?view=vs-2022

[6] Github, “Github copilot now has a better ai model and new capabilities,” 2023. [Online]. Available: https://github.blog/2023-02-14-github-copilot-now-has-a-better-ai-model-and-new-capabilities/

[7] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, “Evaluating large language models trained on code,” *arXiv:2107.03374*, 2021. [Online]. Available: https://arxiv.org/abs/2211.05102

[8] OpenAI, “Humaneval dataset,” 2021. [Online]. Available: https://github.com/openai/human-eval

[9] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt, “Measuring coding challenge competence with apps,” *NeurIPS*, 2021. [Online]. Available: https://openreview.net/forum?id=sD93GOzH3i5

[10] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, and P. Fung, “Survey of hallucination in natural language generation,” *ACM Comput. Surv.*, vol. 55, no. 12, mar 2023. [Online]. Available: https://doi.org/10.1145/3571730

[11] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. de Las Casas, L. A. Hendricks, J. Welbl, A. Clark, T. Hennigan, E. Noland, K. Millican, G. van den Driessche, B. Damoc, A. Guy, S. Osindero, K. Simonyan, E. Elsen, J. W. Rae, O. Vinyals, and L. Sifre, "Training compute-optimal large language models," *arXiv:2203.15556*, 2022. [Online]. Available: https://arxiv.org/abs/2211.05102

[12] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.abq1158

[13] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv:2203.13474*, 2023. [Online]. Available: https://arxiv.org/abs/2211.05102

[14] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," *arXiv:2202.13169*, 2022. [Online]. Available: https://arxiv.org/abs/2211.05102

[15] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W. tau Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," *arXiv:2204.05999*, 2022. [Online]. Available: https://arxiv.org/abs/2211.05102

[16] L. B. Allal, R. Li, D. Kocetkov, C. Mou, C. Akiki, C. M. Ferrandis, N. Muennighoff, M. Mishra, A. Gu, M. Dey, L. K. Umapathi, C. J. Anderson, Y. Zi, J. L. Poirier, H. Schoelkopf, S. Troshin, D. Abulkhanov, M. Romero, M. Lappert, F. D. Toni, B. G. del Río, Q. Liu, S. Bose, U. Bhattacharyya, T. Y. Zhuo, I. Yu, P. Villegas, M. Zocca, S. Mangrulkar, D. Lansky, H. Nguyen, D. Contractor, L. Villa, J. Li, D. Bahdanau, Y. Jernite, S. Hughes, D. Fried, A. Guha, H. de Vries, and L. von Werra, "Santacoder: don't reach for the stars!" *arXiv:2301.03988*, 2023. [Online]. Available: https://arxiv.org/abs/2211.05102

[17] M. Bavarian, H. Jun, N. Tezak, J. Schulman, C. McLeavey, J. Tworek, and M. Chen, "Efficient training of language models to fill in the middle," *arXiv:2207.14255*, 2022. [Online]. Available: https://arxiv.org/abs/2211.05102

[18] C. Donahue, M. Lee, and P. Liang, "Enabling language models to fill in the blanks," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, July 2020. [Online]. Available: https://aclanthology.org/2020.acl-main.225 pp. 2492–2501.

[19] A. Aghajanyan, B. Huang, C. Ross, V. Karpukhin, H. Xu, N. Goyal, D. Okhonko, M. Joshi, G. Ghosh, M. Lewis, and L. Zettlemoyer, "Cm3: A causal masked multimodal model of the internet," *arXiv:2201.07520*, 2022. [Online]. Available: https://arxiv.org/abs/2211.05102

[20] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Comput. Surv.*, vol. 55, no. 9, jan 2023. [Online]. Available: https://doi.org/10.1145/3560815

[21] Y. Ding, Z. Wang, W. U. Ahmad, M. K. Ramanathan, R. Nallapati, P. Bhatia, D. Roth, and B. Xiang, "Cocomic: Code completion by jointly modeling in-file and cross-file context," *arXiv:2212.10007*, 2022. [Online]. Available: https://arxiv.org/abs/2211.05102

[22] F. Zhang, B. Chen, Y. Zhang, J. Liu, D. Zan, Y. Mao, J.-G. Lou, and W. Chen, "Repocoder: Repository-level code completion through iterative retrieval and generation," *arXiv:2303.12570*, 2023. [Online]. Available: https://arxiv.org/abs/2211.05102

[23] D. Shrivastava, H. Larochelle, and D. Tarlow, "Repository-level prompt generation for large language models of code," *arXiv:2206.12839*, 2022. [Online]. Available: https://arxiv.org/abs/2211.05102

[24] P. Thakkar, "Copilot internals," 2022. [Online]. Available: https://thakkarparth007.github.io/copilot-explorer/posts/copilot-internals

[25] J. Xu, X. Liu, J. Yan, D. Cai, H. Li, and J. Li, "Learning to break the loop: Analyzing and mitigating repetitions for neural text generation," in *NeurIPS*, 2022. [Online]. Available: https://arxiv.org/abs/2206.02369

[26] N. Carlini, D. Ippolito, M. Jagielski, K. Lee, F. Tramer, and C. Zhang, "Quantifying memorization across neural language models," in *The Eleventh International Conference on Learning Representations*, 2023. [Online]. Available: https://openreview.net/forum?id=TatRHT_1cK

[27] N. Carlini, F. Tramèr, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. B. Brown, D. X. Song, Ú. Erlingsson, A. Oprea, and C. Raffel, "Extracting training data from large language models," in *USENIX Security Symposium*, 2020. [Online]. Available: https://www.usenix.org/system/files/sec21-carlini-extracting.pdf

[28] Nvidia, "Accelerated inference for large transformer models using nvidia fastertransformer and nvidia triton inference server," 2022. [Online]. Available: https://developer.nvidia.com/blog/accelerated-inference-for-large-transformer-models-using-nvidia-fastertransformer-and-nvidia-triton-inference-server/

[29] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "Gptq: Accurate post-training quantization for generative pre-trained transformers," *arXiv:2210.17323*, 2023. [Online]. Available: https://arxiv.org/abs/2211.05102

[30] L. Lew, V. Feinberg, S. Agrawal, J. Lee, J. Malmaud, L. Wang, P. Dormiani, and R. Pope, "Aqt: Accurate quantized training)," 2022. [Online]. Available: http://github.com/google/aqt

[31] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, "Llm.int8(): 8-bit matrix multiplication for transformers at scale," *arXiv:2208.07339*, 2022. [Online]. Available: https://arxiv.org/abs/2211.05102

[32] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, A. Levskaya, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," *arXiv:2211.05102*, 2022. [Online]. Available: https://arxiv.org/abs/2211.05102

[33] Github, "Github copilot x: The ai-powered developer experience," 2023. [Online]. Available: https://github.blog/2023-03-22-github-copilot-x-the-ai-powered-developer-experience/

[34] Cursor, "Cursor.so home page," 2023. [Online]. Available: https://www.cursor.so/

[35] H. J. Branch, J. R. Cefalu, J. McHugh, L. Hujer, A. Bahl, D. del Castillo Iglesias, R. Heichman, and R. Darwishi, "Evaluating the susceptibility of pre-trained language models via handcrafted adversarial examples," *arXiv:2209.02128*, 2022. [Online]. Available: https://arxiv.org/abs/2211.05102