Kaunas University of Technology

Faculty of Informatics

Large Language Models for OpenAPI Definition Autocompletion

Master's Final Degree Project

Bohdan Petryshyn

Project author

Assoc. Prof. Dr. Mantas Lukoševičius

Supervisor

Kaunas, 2024

Kaunas University of Technology

Faculty of Informatics

Large Language Models for OpenAPI Definition Autocompletion

Master’s Final Degree Project

Artificial Intelligence in Computer Science (6211BX007)

Bohdan Petryshyn

Project author

Assoc. Prof. Dr. Mantas Lukoševičius

Supervisor

Prof. Dr. Alfonsas Misevičius

Reviewer

Kaunas, 2024

Kaunas University of Technology

Faculty of Informatics

Bohdan Petryshyn

Large Language Models for OpenAPI Definition Autocompletion

Declaration of Academic Integrity

I confirm that the final project of mine, Mantas Lukoševičius, on the topic „Large Language Models for OpenAPI Definition Autocompletion“ is written completely by myself; all the provided data and research results are correct and have been obtained honestly. None of the parts of this thesis have been plagiarised from any printed, Internet-based or otherwise recorded sources. All direct and indirect quotations from external resources are indicated in the list of references. No monetary funds (unless required by Law) have been paid to anyone for any contribution to this project.

I fully and completely understand that any discovery of any manifestations/case/facts of dishonesty inevitably results in me incurring a penalty according to the procedure(s) effective at Kaunas University of Technology.

(name and surname filled in by hand) (signature)

Petryshyn, Bohdan. Large Language Models for OpenAPI Definition Autocompletion. Master's Final Degree Project / supervisor assoc. prof. dr. Mantas Lukoševičius; Informatics Faculty, Kaunas University of Technology.

Study field and area (study field group): Computer science, Informatics (B01).

Keywords: Large language models, Code Llama, GitHub Copilot, OpenAPI, code completion, code generation, fine-tuning, prompt engineering, benchmarking.

Kaunas, 2024. 52 p.p.

Summary

Recent advancements in Large Language Models (LLMs) and their utilization in code generation tasks have significantly reshaped the field of software development. Despite the remarkable efficacy of code completion solutions in mainstream programming languages, their performance lags when applied to less ubiquitous formats such as OpenAPI definitions. This study evaluates the OpenAPI completion performance of GitHub Copilot, a prevalent commercial code completion tool, and proposes a set of task-specific optimizations leveraging Meta's open-source model, Code Llama. A semantics-aware OpenAPI completion benchmark proposed in this research is used to perform a series of experiments through which the impact of various prompt engineering and fine-tuning techniques on the Code Llama model's performance is analyzed. The fine-tuned Code Llama model reaches a peak correctness improvement of 55.2% over GitHub Copilot despite utilizing 25 times fewer parameters than the commercial solution's underlying Codex model. Additionally, this research proposes an enhancement to a widely used code infilling training technique, addressing the issue of underperformance when the model is prompted with context sizes smaller than those used during training.

Petryshyn Bohdan. Dideli kalbos modeliai „OpenAPI“ apibrėžimų automatiniam užbaigimui. Magistro baigiamasis projektas / vadovas doc. dr. Mantas Lukoševičius; Kauno technologijos universitetas, Informatikos fakultetas.

Studijų kryptis ir sritis (studijų krypčių grupė): Informatikos mokslai, Informatika (B01).

Reikšminiai žodžiai: Didieji kalbos modeliai, Code Llama, GitHub Copilot, OpenAPI, kodo užbaigimas, kodo generavimas, modelio tikslinantysis mokymas, užklausų inžinerija, lyginamoji analizė.

Kaunas, 2024. 52 p.p.

Santrauka

Naujausi didžiųjų kalbos modelių (angl. *large language model*) pasiekimai ir jų panaudojimas kodo generavimo užduotyse žymiai pakeitė programinės įrangos kūrimo sritį. Nepaisant išskirtinio kodo užbaigimo sprendimų efektyvumo pagrindinėms programavimo kalboms, jų veikimas atsilieka taikant mažiau paplitusiems formatams, tokiems kaip OpenAPI apibrėžimai. Šiame tyrime vertinamas OpenAPI užbaigimo našumas naudojant GitHub Copilot, populiarų komercinį kodo užbaigimo įrankį, ir pasiūloma aibė užduočiai specifinių optimizacijų, pasinaudojant Meta atvirojo kodo modeliu Code Llama. Šiame tyrime pasiūlytas semantikai jautrus OpenAPI užbaigimo lyginamasis standartas (angl. *benchmark*) naudojamas eksperimentų serijai, kurių metu analizuojamas įvairių užklausų inžinerijos (angl. *prompt engineering*) ir modelio tikslinamojo mokymo (angl. *fine-tuning*) metodų poveikis Code Llama modelio našumui. Patikslintas Code Llama modelis pasiekia 55,2% didesnį teisingumą nei GitHub Copilot, nepaisant to, kad naudoja 25 kartus mažiau parametrų nei šio komercinio sprendimo vidinis Codex modelis. Be to, šiame tyrime siūloma patobulinti plačiai naudojamą kodo užpildymo (angl. *code infilling*) mokymo metodiką, sprendžiant sumažėjusio rezultato tikslumo problemą, kai modelis užklausiamas su mažesniu kontekstu, nei kad buvo naudojamas mokymo metu.

Table of Contents

List of Tables

List of Figures

List of Abbreviations and Terms

Abbreviations:

AI – Artificial Intelligence;

API – Application Programming Interface;

AWS - Amazon Web Services;

GPU – Graphics Processing Unit;

IDE – Integrated Development Environment;

LLM – Large Language Model;

NLP – Natural Language Processing;

PEFT – Parameter-Efficient Fine-Tuning;

REST – Representational State Transfer;

RNN – Recurrent Neural Network;

UML – Unified Modeling Language.

Terms:

OpenAPI specification – A standard for defining APIs in a machine and human-readable format.

OpenAPI definition – A document in YAML or JSON format that describes an instance of an API following the OpenAPI specification.

Introduction

In the world of constantly growing digitalization and automation, Application Programming Interfaces (APIs) play a central role in enabling communication between software systems. The OpenAPI specification is a widely adopted standard for defining APIs in both machine and human-readable format. An OpenAPI definition describes all the requests, their parameters, and responses supported by the API. Since the format is machine-readable, a variety of applications were developed to leverage OpenAPI definitions for generating API documentation and client libraries, performing automated testing, or even automatically integrating with the API using the recent advancements in Artificial Intelligence (AI) and Large Language Models (LLMs).

Project Novelty and Relevance

Although OpenAPI definitions can be fully generated based on the existing API implementation, writing a definition manually before implementing the API is a common practice that enables the development of higher-quality APIs. Writing OpenAPI definitions by hand is often a time-consuming and error-prone process due to the verbosity of the format. Modern polyglot code completion solutions like GitHub Copilot and Amazon Web Services (AWS) Code Whisperer have shown that LLMs can be used to increase developer productivity by providing code suggestions based on the surrounding context. These solutions are integrated into ubiquitous Integrated Development Environments (IDEs) and are capable of suggesting high-quality code completions in popular programming languages such as Python, JavaScript, and Java. However, the performance of these solutions in less popular formats like OpenAPI leaves much to be desired.

This research evaluates the OpenAPI completion performance of one of the most widely used commercial code completion solutions, GitHub Copilot, and investigates the feasibility of developing a better-performing solution based on a modern open-source LLM Code Llama. To achieve this, a novel semantics-aware OpenAPI completion benchmark is proposed and used to evaluate the performance of both solutions. An extensive set of experiments is performed to analyze the impact of a variety of variables, prompt engineering, and fine-tuning techniques on the performance of the Code Llama model. The results of the experiments reveal that the original version of the model, when used with optimal prompt format is already capable of outperforming GitHub Copilot in OpenAPI completion task. The fine-tuned version of the model further widens the performance gap with the commercial solution. To the best of the authors' knowledge, this is the first work investigating the Code Llama fine-tuning for code infilling in detail. The OpenAPI completion benchmark, the fine-tuning pipeline, and the cross-platform code completion JavaScript library were made publicly available on GitHub [1, 2, 3].

Aim and Objectives

The aim of this research is to investigate the approaches for developing and optimizing an OpenAPI completion solution that would outperform the existing commercial or open-source solutions in this task. The objectives of the research are as follows:

1. To analyze the state of the art in code generation and evaluation and identify the challenges in this area in the context of OpenAPI completion.
2. To identify the baseline solution for OpenAPI completion as well as the foundational model for the study.
3. To design and implement a semantics-aware benchmark for OpenAPI completion evaluation.
4. To design and implement a reusable integrated OpenAPI completion solution based on the foundational model.
5. To propose a set of prompt engineering and fine-tuning techniques for optimizing the foundational model's performance in OpenAPI completion.
6. To evaluate the OpenAPI completion performance of the proposed solution, compare it to the baseline using the developed benchmark and analyze the impact of the hypothesized techniques on the model's performance.

Document Structure

In the introduction chapter, the relevance and the novelty of the research are discussed as well as the aim and the intermediate objectives of the study are defined. Chapter 1 provides an overview of the current state of the art in code generation, including the transformer-based models, relevant approaches to code infilling, performance optimization and evaluation methods, and integrated code generation solutions that are widely used in the industry. Chapter 2 describes the novel methods proposed in this research, including the semantics-aware OpenAPI completion benchmark, the foundational model selection, the task-specific prompt engineering and fine-tuning techniques, and the architecture of the reusable integrated OpenAPI completion solution. Chapter 3 presents the results of the experiments performed in the study, including the evaluation of the commercial baseline solution and the proposed open-source LLM-based solution and the impact of the prompt engineering and fine-tuning techniques on the foundational model's performance. Finally, the conclusion chapter summarizes the research findings and implications.

Previous Publications

At the time of writing, part of this research is being submitted for publication in a peer-reviewed scientific journal. The preprint of the article is available on Arxiv [4].

1. Overview of the State of the Art in Code Generation

As technology advances, the landscape of code generation continues to evolve, with transformer-based models and integrated solutions revolutionizing the field. This section delves into the state-of-the-art solutions in code generation, explores integrated solutions that streamline the development process, assesses methodologies for evaluating the quality of generated code, and highlights the challenges in OpenAPI autocompletion as a specific case of code generation.

1.1. Transformer-based Code Generation Models

Transformer-based models are the most widely used type of models for Natural Language Processing (NLP) and code generation in particular [5]. The original transformer model was introduced in the famous ”Attention Is All You Need” article in 2017 [6]. The model was based on the trainable neural attention mechanism proposed by Bahdanau Dzmitry et al. [7] earlier in 2014. This mechanism presented a solution to several problems of Recurrent Neural Network (RNN) based models that were one of the primary methods for NLP at the time [8].

One of the main shortcomings of RNN-based models is the inability to capture long-term dependencies in the input due to vanishing and exploding gradient problems and limited hidden state size [9]. The attention mechanism allows the model to capture dependencies between any two tokens in the input sequence, drawing the likelihood of a long-term dependency being captured the same as the likelihood of any short-term dependency.

Another major drawback of RNN-based models is the sequential nature of processing which makes it impossible to efficiently parallelize the computation. A significant part of the attention computation can be done independently for each token in the input sequence which dramatically speeds up processing on modern highly parallel hardware like Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs).

Transformer model architecture can consist of an encoder, decoder, or a combination of both (Figure 1). Each of these parts leverages the attention mechanism in different ways to achieve its objectives. The role of an encoder is to analyze the input sequence and encode it into an internal representation that captures dependencies between the input tokens. This is done using a mechanism called self-attention where the attention score is calculated between each pair of the input tokens and a new value is calculated for each token as a sum of all the values in the input sequence weighted by the attention score. The role of a decoder is to autoregressively generate the output sequence leveraging a mechanism called encoder-decoder attention. In this process, each output token is generated as an attention-weighted sum of the encoded input tokens and the previously generated output tokens.

The encoder-only models like BERT from Google [10], RoBERTa [11], AlBERT [12] and DeBERTa [13] only consist of an encoder and do not produce sequences as the output. Instead, these models only produce encoded representations of the input sequence that can be used for tasks like input classification, sentiment analysis, or feature extraction. The objective used during training is often not well-aligned with the inference objective, so these models usually

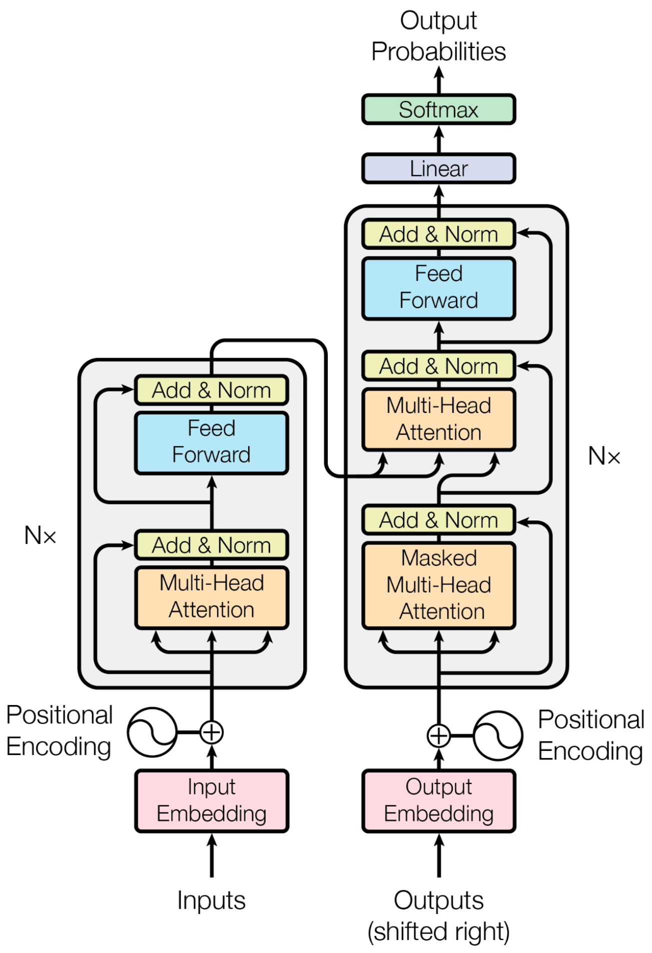


Fig. 1. Encoder-decoder transformer model architecture. Source: [6]

require fine-tuning on the specific target task.

The encoder-decoder models like the original transformer suggested by Ashish Vaswani et al., T5 [14] and BART from Meta [15] incorporate both encoder and decoder in their architecture. These models are capable of sequence-to-sequence (seq-to-seq) generation and are commonly used for machine translation and summarization where the input and the expected output sequences can be clearly defined.

The decoder-only models like the Generative Pre-trained Transformer (GPT) family of models from OpenAI [16], PaLM from Google [17], Chinchilla from Deepmind [18], and Llama from Meta [19, 20] only consist of a decoder. This preserves the ability to analyze the input sequence and generate a sequence as the output but also eliminates the boundary between the input and the output. The latter characteristic makes this type of models more generic and often abolishes the need for fine-tuning after the initial training as most of the tasks can be represented as a natural language seq-to-seq function [21]. Compared to the encoder-decoder architectures, which are also capable of seq-to-seq generation, decoder-only architectures further streamline the training process by removing the need to define the input and output formats for the model to operate on. These models are often used for language modeling and text generation but are also capable of input classification, sentiment analysis, and machine translation if invoked with a corresponding prompt. The capabilities of the large decoder-only models can be further expanded with prompting techniques like few-shot prompting [22].

Code generation is a broad term that is used to describe a wide range of activities. One such activity is code solution generation for a natural language problem definition. This task is usually approached as a seq-to-seq generation problem where the task definition is the input, and the code solution is the output of the model. One of the successful solutions in this area is AlphaCode published by DeepMind [23]. The model leverages the encoder-decoder architecture. It is pre-trained on a variety of public GitHub repositories in different programming languages and fine-tuned on a curated dataset of competitive programming tasks. As stated by the authors, the model achieved a ranking of top 54.3% among more than 5 000 participants on the Codeforces platform.

Code generation from a natural language task definition might be useful for no-code or low-code engineers but code completion, where the model generates continuation for an incomplete code input, represents everyday programmers' work much closer. Models like Codex from OpenAI [24], CodeGen from Salesforce [25, 26], and Code Llama from Meta [27] leverage decoder-only architectures for autoregressive code modeling. The fill-in-the-middle (also referred to as infilling) transformation and training objective allow decoder-only models to include the context from both sides of the generated part, removing the limitation of strict left-to-right generation while maintaining the ability to output long sequences of tokens [28]. This approach not only enables context-aware code completion but also supports natural language task definition to some degree. These models can be prompted with a natural language code comment to generate an implementation. This can be explained by a weak natural language signal that might be captured during training on large corpora of code with comments and code-related discussions [25].

1.2. Transformer Model Performance Optimization

Training LLMs from scratch is a highly computationally-intensive process [18]. A more practical approach to LLM application is optimizing a so-called foundational model, pre-trained for a similar function, to the specific task.

Prompt engineering and model fine-tuning are two primary methods for LLM performance optimization. A common workflow for improving the performance of a transformer-based model in a specific domain is to use prompt engineering methods to achieve the best possible performance with the base model and then fine-tune the model on a task-specific dataset if further performance improvement is required. This research will utilize both prompt engineering and fine-tuning to improve the performance of the Code Llama model on the OpenAPI definitions generation task.

Prompt engineering is based on the idea that the model can be guided to generate more accurate output by optimizing the prompt in a series of human-insight-driven experiments. [29]. Common prompt engineering techniques for instruction-tuned models include few-shot prompting, chain-of-thought prompting [30], and generated knowledge prompting [31]. For autoregressive code generation, prompt engineering techniques include embedding additional context to the prompt in the form of comments or other language-specific constructs [25] and optimizing the prompt structure and size to maximize the model's performance and resource efficiency.

Fine-tuning is the second primary method for improving the performance of LLMs. The idea behind this method is that the model that has already been trained for a similar or a more broad task can be further fine-tuned for a specific task. The fine-tuning process is similar to training, but it is performed on a pre-trained model and requires significantly less data and computational resources. Two fine-tuning approaches are commonly used: full fine-tuning and Parameter-Efficient Fine-Tuning (PEFT).

Full fine-tuning is performed by continuing the training process on an already pre-trained model. During this process, all the model parameters are updated just like in the original training. This means that the hardware requirements for the full fine-tuning are also similar. This approach demonstrates the potential for achieving the best results, but it also requires the most computational resources and data. The ability to change all the model parameters during full fine-tuning can also lead to catastrophic forgetting, a training failure mode where the model loses the ability to perform the original task or stops generating adequate output altogether.

PEFT, as an alternative to full fine-tuning, is performed by freezing most of the pre-trained model parameters and only training a relatively small number of additional parameters. This approach requires much less computational resources and tends to yield comparable results to the full-finetuning [32]. The PEFT approach is also considered to be less prone to catastrophic forgetting as the majority of the pre-trained model parameters are left unchanged during the fine-tuning process. Notable PEFT techniques include adapter tuning [33] which inserts additional adapter layers between the original model layers and only updates them during training, BitFit [34] which only trains the bias-terms of the original model, and Low-Rank Adaptation (LoRA) [35] which decomposes the original parameters into low-rank matrices and only trains them. In Figure 2, A and B are the low-rank adaptation matrices for the original weights matrix W. Only A and B matrices are trained and the total number of parameters in these matrices is $2rd$ (instead of d^2) where r is the rank of the decomposition and d is the size of the input vector x. The LoRA method claims to reduce the number of trainable parameters by 10,000 times and memory requirements by 3 times while maintaining or surpassing the performance of full fine-tuning when applied to models like GPT-3 and RoBERTa.

When it comes to the data used for fine-tuning, the best results are achieved when the fine-tuning data is similar in format to the data used during the pre-training but presents new information. The authors of the Code Llama model also demonstrated that fine-tuning already-seen data can still lead to significant improvement in task-specific performance [27].

1.3. Integrated Code Generation Solutions

Integrated code generation and completion solutions are end-to-end systems that combine code generation models, prompt engineering techniques, developer experience optimizations, and IDE

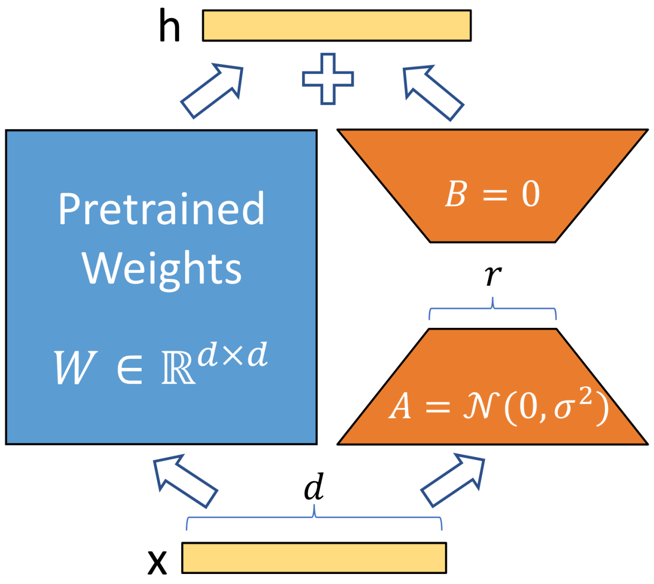

h
Pretrained Weights
$W \in \mathbb{R}^{d \times d}$
$B = 0$
r
$A = \mathcal{N}(0, \sigma^2)$
d
x

Fig. 2. LoRA parametrization. Source: [35]

integrations with the aim of solving a real-world problem of improving developer productivity. These solutions are often proprietary, so the details of their implementation are not available to the public. However, some solutions are open-source and can be analyzed in detail.

The most notable commercial solution is GitHub Copilot [36]. Developed by GitHub, OpenAI, and Microsoft, this code assistant is based on the Codex model. The solution is integrated into several IDEs like Visual Studio Code, NeoVim, and JetBrains IDEs. As discovered by a reverse engineering effort [37], Copilot's prompt-building approach is based on several language-specific heuristics. For example, for programming languages like Python, a list of imported libraries and declared functions within the file is extracted and always included in the prompt. The code assistant underwent manual evaluation in a number of studies [38, 39, 40, 41] demonstrating decent performance in benchmarks like HumanEval [24] and solving moderate complexity tasks at platforms like LeetCode. No studies were found that would evaluate the solution performance in OpenAPI completion. In this research, GitHub Copilot is evaluated as the baseline solution for the OpenAPI definitions completion task.

Other proprietary solutions worth mentioning are Tabnine [42] which accents the security of the user's data and the code base, AWS Code Whisperer [43] which demonstrates additional expertise in AWS services and client libraries and performs additional checks for license compliance and common security vulnerabilities in the generated code, and Sourcegraph Cody [44] which demonstrates awareness of the whole user's code base.

Continue [45] is an example of an open-source integrated code completion solution that aims to be model-agnostic and supports multiple models with the most commonly used being Code Llama Instruct running in a local or a cloud-based LLM server. Other open-source models supported by the solution include Wizard Coder [46], Phind Code Llama [47], Mistral [48], Code Up [49], and Zephyr [50]. Commercial models like GPT-3, GPT-4, and PaLM-2 [51] can be integrated as well. During code assessment, it was discovered that Continue prompts the model in an instructional manner by providing it with the surrounding code and asking to generate the missing part. This explains the fact that the solution only works with models that support instruction execution. Another open-source solution, LLM-vscode [52], is a Visual Studio Code extension developed by Hugging Face for demonstrational purposes. The extension can be used with many models but in contrast to Continue, it supports both instruction execution and code infilling models. LLM-vscode is usually used with models like Code Llama, Phind Code Llama, and WizardCoder.

Another research on applying LLMs to a real-world code completion use case was published by Microsoft in 2020 [53]. Apart from evaluating transformer optimization techniques for code like applying tokenization methods that can overcome the closed vocabulary problem, this research describes multiple developer experience optimization techniques like client-side tree-based caching, parallel beam search decoder, and compute graph optimizations.

1.4. Code Generation Evaluation

Evaluation of LLMs has always been a challenging task in the field of NLP. The evaluation methods have been evolving in step with the abilities of the models and the complexity of the tasks they perform.

As the first applications of language models started appearing in the field of natural language modeling and machine translation, the evaluation methods were designed to measure the quality of the generated text by comparing it to the expected output. The most widely used metrics for this purpose are BLEU [54], ROUGE [55], and METEOR [56]. These metrics are based on calculating the similarity by counting the overlapping n-grams or measuring the longest common sequences.

Code generation can not be reliably evaluated with machine translation metrics due to the fundamental differences between natural language and code. The code is structured and possesses a more rigid syntax than natural language, making it more sensitive to small character-level changes. Another source of complexity in code evaluation is the fact that a single code problem can have multiple equally correct solutions. This makes it impossible to define an exhaustive set of expected outputs for a given input.

One of the current state-of-the-art evaluation methods for code generation is the HumanEval benchmark. Originally proposed by OpenAI, this benchmark was designed to evaluate Python code generation by providing a set of problems each with a description and a set of automated test cases. Later, the method was extended to support 18 additional programming languages by the MultiPL-E benchmark [57]. These methods take language semantics into account but are only applicable to a limited set of programming languages and do not support specific code semantics like OpenAPI definitions.

The OpenAPI specification is not a programming language but a data format specification. This means that the existing code generation evaluation methods can not be easily extended to support OpenAPI completion evaluation. To enable the development of OpenAPI completion solutions, a new benchmark has to be proposed that supports the specific semantics of OpenAPI definitions.

1.5. Open-source Code Generation Models

Open-source LLMs are widely used by researchers and practitioners as foundational models for further optimization and fine-tuning. At the time of writing, the most notable open-source code generation models are DeepSeekCoder [58], Code Llama [27], and Star Coder [59]. These models are based on the decoder-only transformer architecture and are trained with the infilling objective to generate code completions. The models are available in multiple sizes starting from 3 billion parameters and up to 33 billion parameters. As can be seen from Table 1, the performance of the models varies across different programming languages with a tendency to perform better in common languages like Python and JavaScript and worse in less common languages like R.

Table 1. Open-source code generation and infilling LLM performance comparison. Collected from [60, 58]

Model	Python, %	JavaScript, %	Java, %	Rust, %	R, %
DeepSeek Coder 33B	52.45	51.28	43.77	43.78	26.76
DeepSeek Coder 7B	45.83	45.9	37.72	34.67	28.99
Code Llama 13B	35.07	38.26	32.23	29.72	18.32
Code Llama 13B Python	42.89	40.66	33.56	29.32	18.35
Code Llama 7B	29.98	31.8	29.2	25.82	18.04
Code Llama 7B Python	40.48	36.34	29.15	26.96	18.25
Star Coder 15B	30.35	31.7	28.53	24.46	10.18
Star Coder 7B	28.37	27.35	24.44	22.6	14.51
Star Coder 3B	21.5	21.32	19.25	16.32	10.1
Code Gen 2.5 7B Multi	28.7	26.27	26.01	21.84	11.59
Code Gen 2.5 7B Mono	33.08	23.22	19.75	7.83	4.41

1.6. Analysis Conclusion

Based on the analysis performed, the following conclusions can be made:

1. Transformer models are the most widely used machine learning architecture for NLP and code generation. From the three main variances of the transformer architecture, the decoder-only approach turns out to be the most suitable for text and code generation and infilling.
2. The autoregressive nature of decoder-only transformers makes it impossible to capture context from both sides of the generated part. The autoregressive fill-in-the-middle approach suggested by Bavarian et al. [28] enables infilling capabilities in decoder-only models with minimal performance penalty for the primary autoregressive generation task. The resulting autoregressive infilling models are capable of generating much longer sequences of tokens than the encoder or encoder-decoder models.
3. Foundational models like Code Llama or CodeGen provide a more optimal pipeline for natural language problem solving. Fine-tuning a pre-trained LLM is much more cost and resource-efficient than training a model from scratch. The fine-tuning process can be further optimized with PEFT methods like LoRA.

4. Fine-tuning LLMs with data already seen during training still leads to significant performance improvements as proved by the researchers behind the Code Llama model.
5. Integrated code generation solutions involve not only LLMs but also integration layers consisting of the pre-processing and post-processing steps. Human insight-driven prompt engineering is a major contributor to the resulting performance of end-to-end solutions.
6. Code generation performance cannot be evaluated using common natural language generation metrics and requires semantics reasoning and evaluation. Programming languages are often evaluated by executing a set of automated tests on the generated code snippet.

The following areas of improvement in the field of OpenAPI code completion were identified:

1. Existing generic code completion solutions demonstrate good performance in common languages like Python and JavaScript. However, their performance in more narrow-purpose languages like R or OpenAPI is usually lower than satisfactory.
2. No evaluation framework exists for OpenAPI completion. Developing such a framework would allow the evaluation of existing code generation solutions in OpenAPI completion as well as stimulate the development of the new OpenAPI-specific code completion solutions.

2. Methods and Datasets for OpenAPI Completion

As the previous section outlined, OpenAPI completion requires specialized evaluation and optimization methods. This section describes the proposed OpenAPI completion benchmark and the corresponding dataset, the foundational model selected for the study, and the prompt engineering techniques used to optimize the model's performance.

2.1. OpenAPI Completion Evaluation Criteria and Benchmarking

As discussed in the previous section, existing text or code generation evaluation methods can not be considered optimal for OpenAPI generation evaluation. These methods are not semantics-aware as in the case with machine language translation metrics or do not support specific OpenAPI semantics as in the case with the widely used HumanEval and MultiPL-E benchmarks.

The proposed benchmark closely resembles the real-world usage of the solution in terms of input modeling and output evaluation. Figure 3 outlines the steps for a single evaluation iteration. The masking step, where part of the input definition is replaced with a special mask token, is responsible for modeling the input. At the completion step, the evaluated model is used to replace the mask token with the generated OpenAPI definition. The evaluation step, where the completed definition is compared to the original, is responsible for the completion result evaluation.

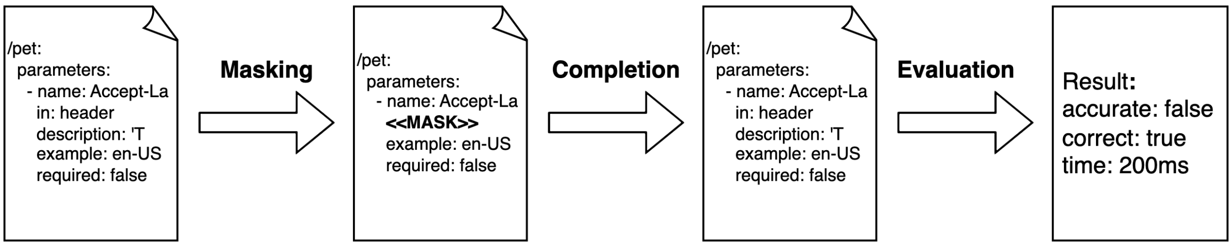


Fig. 3. Single evaluation iteration

The masking step can be seen as splitting the original OpenAPI definition into two parts (prefix and suffix) and removing some amount of the definition from between. This way, the input modeling is defined by the position of the prefix-suffix split and the size of the masked part between them.

When editing an OpenAPI definition, the user does not always add new endpoints or parameters and can expect to get a completion suggestion even when they have not finished typing the previous word or syntactic construct. This means that the prefix and suffix split should be randomly selected and not tied to a new line or a keyword. Conversely, the end of the masked sequence should be on a new line, as it is reasonable to assume that users are more likely to edit a document at the end of a line rather than in the middle of it when describing new API operations or adding parameters to existing ones. An example of a masked OpenAPI definition is available in Appendix 2.

The following metrics are measured at the evaluation step:

1. Correctness – how often the generated OpenAPI definition is semantically identical to the

original one.

2. Validity – how often the generated OpenAPI definition is syntactically valid.
3. Speed – how fast the solution generates the OpenAPI definition.

To be semantically identical, two snippets of OpenAPI definition do not have to be identical at the character level. The YAML and JSON specifications allow for different ways of representing the same data. For example, keys can be optionally quoted, and the order of keys is not significant. The OpenAPI specification allows for even more flexibility with the support of references and other advanced syntactic features. Additionally, OpenAPI definitions contain subjective information such as descriptions, titles, summaries, and examples of data. These fields are not required to be identical for the generated OpenAPI definition to be considered semantically identical to the expected one.

To evaluate correctness, the benchmark uses the Oasdiff tool [61] which is capable of calculating the semantic difference between two OpenAPI definitions. The calculated difference is not affected by the YAML, JSON, or OpenAPI specification flexibility described above. To tolerate subjectiveness in the definition, the following set of heuristics is applied to the calculated difference:

1. The definitions are semantically identical if the calculated difference is empty.
2. The definitions are semantically identical if the calculated difference only contains insignificant changes.
3. Changes in the following subjective fields are considered insignificant: description, summary, and title.
4. Changes in the examples of data are considered insignificant if they do not change the structure and type of the data.

2.1.1. Evaluation Dataset

The dataset for the benchmark was collected from the APIs-guru definitions directory [62] which is a weekly updated collection of public OpenAPI definitions. The directory contains more than 4 000 definitions in YAML format.

Ideally, the definitions have to be unseen by the model during training or fine-tuning. However, as the Code Llama model authors do not reveal the details of the training dataset, it is not possible to guarantee that the definitions were not used during training. Only the newest definitions in the directory were selected for the evaluation dataset to minimize the chance of being seen by the model. The definitions were also checked for presence in The Stack [63] dataset which is one of the biggest open code datasets collected from GitHub. The definitions that were present in The Stack dataset were excluded from the evaluation dataset.

Another requirement for the definitions in the evaluation dataset is that they have to represent different domains and styles. The APIs-guru collection contains definitions for APIs from a variety of domains such as payment systems, cloud services, and electronic commerce. The directory also contains definitions from many producers such as Amazon, Meta, and Google. The definitions were selected from different domains and producers to ensure that the benchmark is not biased toward a specific field or style of OpenAPI definitions. Automatically generated definitions were detected and excluded from the evaluation dataset as they do not represent the future real-world use of the solution.

Only OpenAPI definitions larger than 3 000 lines were selected for the evaluation dataset with the preference given to definitions larger than 20 000 lines when all the other criteria are equal. Large definitions are preferred since they are more likely to contain complex syntactic constructs and semantic dependencies. Small definitions that fit into the context size of the model are often too trivial to complete correctly and do not present any practical value for the benchmark.

Ten OpenAPI definitions satisfying the above criteria were selected for the evaluation dataset (Table 2). Considering the size of the selected definitions, multiple test cases could be extracted from each definition. The test cases were extracted by randomly selecting a position in the definition and masking the rest of the line and ten following lines. The evaluation objective was to generate the infilling for the masked part so that the resulting definition is semantically identical to the original one.

Table 2. OpenAPI definitions included in the evaluation dataset

API Title	Producer	Domain	Size, lines
Adyen Management API	Adyen	Payment system	20 359
Amazon VPC Lattice	AWS	Cloud computing	6 709
Visma e-conomic OpenAPI	Visma	Accounting system	10 162
Google Display & Video 360 API	Google	Video hosting	16 008
Mux API	Mux	Video hosting	7 128
Rubrik REST API	Rubrik	Data security	43 540
Sinao API	Sinao	Accounting system	12 215
SpaceTraders API	SpaceTraders	Gaming	3 121
The Racing API	The Racing API	Gambling	9 650
Severa Public Rest API Documentation	Visma	Project management	36 871

The benchmark was implemented with the idea of being fully automated and reproducible. This made it possible to easily evaluate the performance of many solutions and configurations. However, the fact that commercial solutions like GitHub Copilot are only available as IDE plugins and require manual interaction with the IDE imposed a limitation on the size of the evaluation dataset. To keep the manual evaluation time reasonable while still providing a representative evaluation, the dataset size was constrained to 100 test cases in total which is comparable to the size of the HumanEval benchmark with 164 code problems.

2.2. Platforms and Tools

For the solution implementation, it was decided to use Code Llama as the foundation LLM, Hugging Face Inference Endpoints as the infrastructure platform, Hugginf Face Transformers and PEFT as the fine-tuning libraries, JavaScript as the programming language for the prototype implementation, and Python as a general purpose scripting language for the model fine-tuning. In this section, the key factors that influenced the decisions are discussed.

2.2.1. Foundational Model

Training LLMs from scratch is a highly computationally expensive process [18]. For practical reasons, it is often more efficient to incorporate a generic pre-trained model that can be fine-tuned for a specific task or used in a zero-shot manner. For the OpenAPI completion task, the foundation model should satisfy the following requirements:

1. The model should be available for research and commercial use under a permissive license.
2. The model should be trained on a wide range of programming languages.
3. The model should support large context (at least 4096 tokens) and long output sequences (at least 1024 tokens).
4. The model should support code infilling and autoregressive generation.
5. The model should support fine-tuning.
6. The model should have a big community and be well-documented.

The Code Llama model [27] was selected as the most widely used model that satisfies all the requirements. This code generation model from Meta is trained on an unbound set of programming languages which not only ensures its ability to generate OpenAPI definitions but also allows it to generate code samples that are commonly found in OpenAPI definitions in a variety of programming languages. The model supports large context sizes (up to 100 000 tokens) which is important for the OpenAPI completion task as the definitions can reach tens of thousands of lines in size.

This code generation model is based on another widely adopted natural language model from the same company – Llama 2. The model was trained for code infilling on an additional 500 billion tokens of code. Figure 4 outlines the training pipeline of the Code Llama model family. The model was further trained for Python specialization and fine-tuned for instruction execution, but for the sake of this research, the base Code Llama model is used.

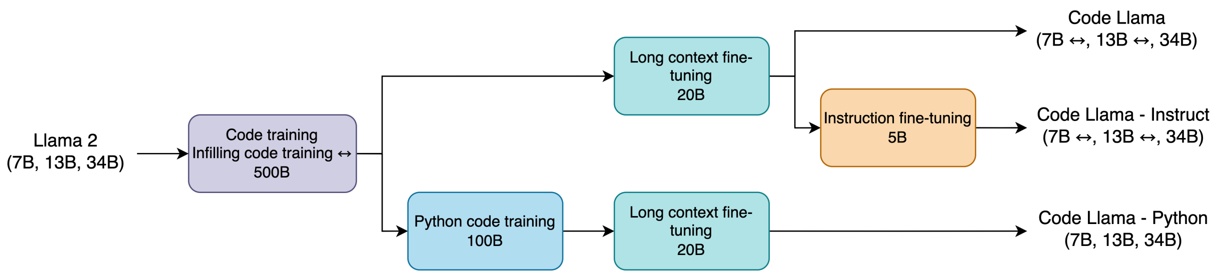


Fig. 4. Code Llama family training pipeline. Adapted from: [27]

The model is available in three sizes: 7 billion parameters, 13 billion parameters, and 34 billion parameters. In this research, the 7 billion and 13 billion parameter models are evaluated. The 34 billion parameter model was not selected for evaluation due to the high computational cost and the lack of evidence that the model would demonstrate significantly higher performance than the 13 billion parameter model. Moreover, the largest model was not trained on code infilling tasks, which is required for real-world code completion scenarios.

The Code Llama model can operate in two modes: autoregressive code generation, where the model takes a code snippet and generates a continuation for it token by token, and code infilling, where the model takes two snippets of code and generates the completion that would fill the gap between them. The mode of operation is determined by the format of the prompt. For code generation, the model should be prompted with the code snippet as is. For code infilling, the two snippets (prefix and suffix) should be formatted in a special way that was defined during the model training.

Following the training methods originally described in the work of Bavarian et al. [28], the Code Llama model supports two prompt formats for code infilling: Prefix Suffix Middle (PSM) and Suffix Prefix Middle (SPM). The study has shown that the SPM format produces slightly better results even when the model is trained on the PSM format. The SPM format is also expected to result in a higher key-value cache hit rate as the suffix is less likely to change in a real-world code infilling scenario. Both formats are evaluated in this research.

The Code Llama model introduces four special tokens to the existing Llama 2 tokenizer: <PRE>, <SUF>, <MID>, <EOT>. Using these tokens, the PSM prompt format can be defined as

$$\text{<PRE>} \circ \text{Enc(prefix)} \circ \text{<SUF>} \circ \text{Enc(suffix)} \circ \text{<MID>} \circ \text{Enc(middle)}, \tag{1}$$

Source: [28]

where the $\text{Enc}(\cdot)$ is the encoding operation and $\circ$ is a token-level concatenation. The SPM format can be defined as

$$\text{<PRE>} \circ \text{<SUF>} \circ \text{Enc(suffix)} \circ \text{<MID>} \circ \text{Enc(prefix)} \circ \text{Enc(middle)}. \tag{2}$$

Source: [28]

The model autoregressively generates tokens until the <EOT> token is generated indicating a successful connection of the middle part with the suffix. Please, note that in the SPM format, the middle part immediately follows the prefix which makes the format even more similar to the simple regression case.

2.2.2. Infrastructure Platform

Running inference, training, and fine-tuning LLMs imposes significant resource requirements beyond the capabilities of a consumer-grade computer. Applying the solution to a real-world problem in a commercial setting requires a scalable and cost-efficient infrastructure solution. The following requirements were considered when selecting the infrastructure platform:

1. The platform should support cost-efficient per-request or per-minute pricing.
2. The platform should automate the model deployment process.
3. The platform should support simple inference APIs.
4. The platform should support a wide range of hardware options including GPUs and TPUs.
5. The platform should support automatic scaling and load balancing.
6. The platform should have a big community and be well-documented.

Hugging Face was selected as the most widely used platform with the lowest cost and decent documentation. The platform supports the deployment of existing models from the Hugging Face model hub as well as models, trained or fine-tuned with the Hugging Face Transformers [64] and PEFT [65] libraries, among others. The platform provides convenient solutions for both inference and training on a wide range of hardware options.

The Hugging Face Inference Endpoints platform [66] allows automatically deploying transformer-based models as Representational State Transfer (REST) APIs. The solution will be beneficial for the research and prototype implementation as it allows deploying models of different sizes without the need for manual infrastructure provisioning or configuration. This will enable quick evaluation and iteration of the solution. The platform also supports automatic scaling and load balancing, request batching, and caching, significantly improving the solution's performance and cost-efficiency in real-world scenarios.

2.2.3. Client-side Programming Language

The LLM and the infrastructure platform make up the solution's back end. An equally important part of the solution is the client-side integration layer that gathers the user input, constructs the prompt for the model, and processes the model output. To ease the future prototype to production transition, the client-side programming language should be compatible with the target platforms (IDE plugin systems and web browsers). Due to the factors described, contrary to the common practice of using Python for machine learning tasks, JavaScript was selected as the primary programming language for the client-side integration layer.

To format an optimal prompt, the client-side integration layer should be able to estimate the token length of the textual data it operates on. The Hugging Face Transformers library [31, 64] implements a tokenizer for Python programming language but not for JavaScript. A third-party Llama 2 tokenizer implementation for JavaScript is available [67]. As the Code Llama model is based on Llama 2, the tokenizer should be compatible with the model.

As the evaluation framework has to be able to evaluate the end-to-end solution performance, it should be integrated with the client-side layer. For this reason, the evaluation framework should be implemented in JavaScript as well. As the runtime environment for the evaluation framework, NodeJS was selected as it allows running JavaScript code outside of the browser.

2.3. Integrated Solution Architecture

The sequence Unified Modeling Language (UML) diagram (Figure 5) depicts the key parts of the solution architecture as well as their interaction. The prompt builder is responsible for generating the prompt for the infilling model from the cursor context provided by the IDE. The formatted prompt is then submitted to the infilling model which is responsible for infilling the missing part between the prefix and the suffix. The infilling result is then passed to the response embedder which is responsible for merging the result with the cursor context. The response embedder also indicates if the completion is finished. If not, a new prompt is generated from the previous cursor context and the infilling generated in the previous step. The generation process loops until the context is infilled or the maximum generation length is reached.

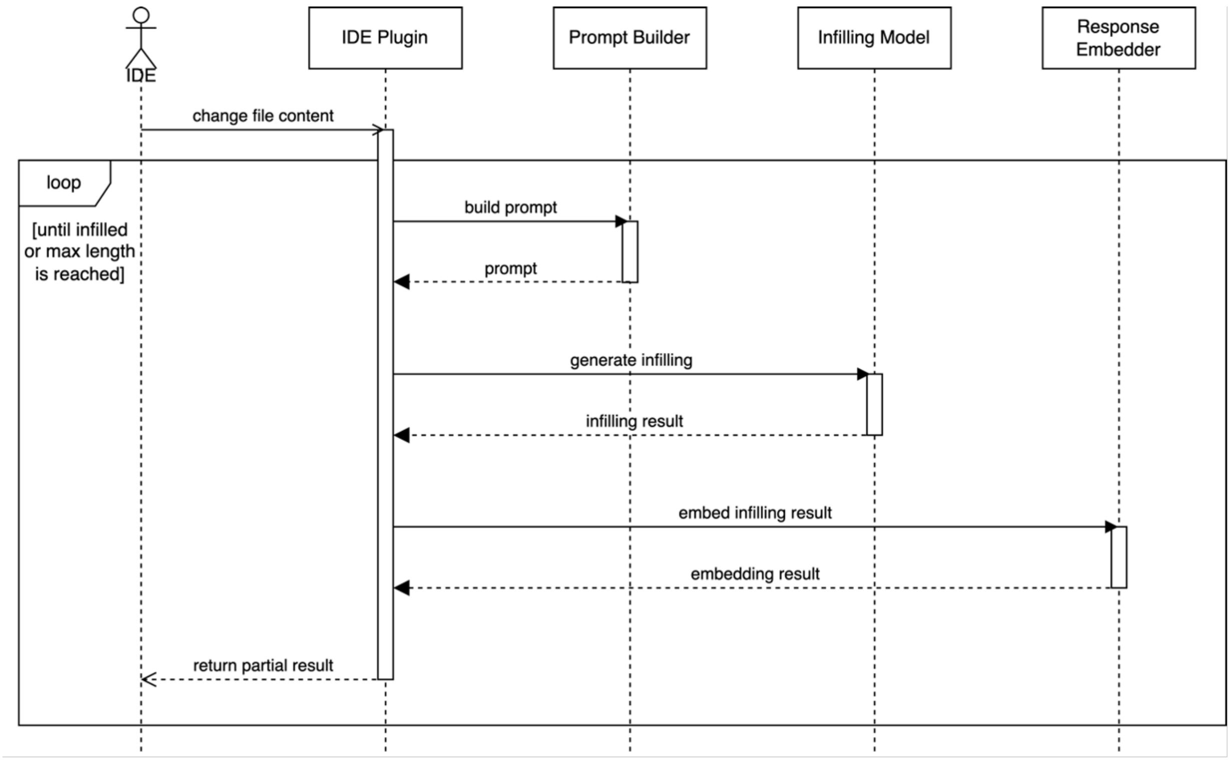


Fig. 5. Integrated code completion solution UML sequence diagram

The combination of the prompt builder, infilling model, and the response embedder make the reusable part of the solution. If packaged as a JavaScript library, such a package could be reused in any IDE that supports JavaScript plugins or any web-based code editor.

Figure 6 outlines the key steps of the response embedder. The response embedder takes the infilling result and the prompt suffix and returns the embedded result that is free from possible overlap with the suffix. The response embedder also indicates if the completion is finished.

2.4. Prompt Engineering Techniques

As the expected OpenAPI definition size is significantly larger than the maximum context size supported by Code Llama, it is not possible to fit the whole definition in the prompt. In this research, the impact of multiple variables on the completion performance is evaluated with the aim of finding the optimal prompt format for a real-world OpenAPI completion task.

One of the factors affecting completion performance is the prefix-to-suffix ratio. Theoretically, prefixes and suffixes should contain the same amount of useful information for the model to generate a semantically correct completion. However, the model can have a preference for one of the parts inherited from the training data. The optimal ratio can be found by evaluating the completion performance for a range of ratios.

Another factor that affects both the completion accuracy and speed is the context size. The context size is the number of tokens available to the model at the time of completion. A larger context size allows the model to take into consideration more information from the surrounding context but also requires more computational time and resources to process. The optimal context size can be found by evaluating the completion performance for a range of context sizes while keeping

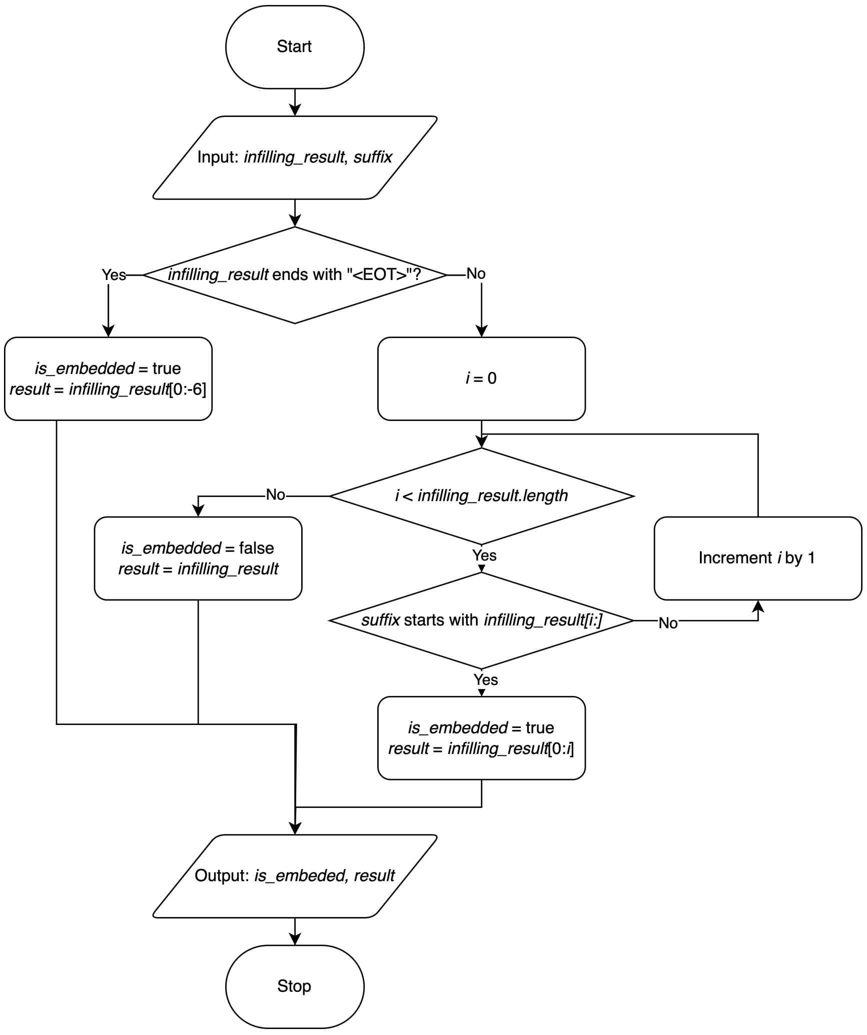

Start
Input: *infilling_result, suffix*
infilling_result ends with "<EOT>"?
Yes
No
is_embedded = true
result = *infilling_result*[0:-6]
i = 0
i < *infilling_result.length*
No
Yes
is_embedded = false
result = *infilling_result*
Increment *i* by 1
suffix starts with *infilling_result[i:]*
No
Yes
is_embedded = true
result = *infilling_result*[0:*i*]
Output: *is_embeded, result*
Stop

Fig. 6. Response embedder UML flow diagram

the prefix-to-suffix ratio constant.

The code infilling mode is expected to be more suitable for the OpenAPI completion task as it allows the model to consider the context both before and after the cursor position. The part of the definition that goes after the cursor should allow the model to take into account components such as schemas, requests, and responses that are commonly placed at the end of the document. The suffix is also important for the model to generate syntactically correct definitions that are integral to the rest of the document.

The autoregressive code generation mode of the Code Llama model can be considered a special case of the code infilling mode with the 100/0 prefix-to-suffix ratio. This mode is not expected to perform on par with the code infilling mode. However, it might still demonstrate better performance than the code infilling mode with a suboptimal prefix-to-suffix ratio as the model can be more focused on the completion task.

OpenAPI specification allows including metadata about the API in the definition. This metadata can include information like the API's title, description, version, contact information, or license type. The metadata is not required for the definition to be considered valid and can be neglected during OpenAPI validation or comparison. However, the metadata may contain useful information for the model to generate a more semantically correct definition.

The components section of the OpenAPI definition is crucial for the model to generate valid definitions since it defines the set of available schemas, requests, responses, and other components that can be referenced in the definition. To test this hypothesis, the prompt builder is modified to extract the components section if available, format it as a comment with the list of component names, and include it in the prompt prefix.

2.5. Code Llama Fine-tuning

The OpenAPI completion performance of the model can be further improved by fine-tuning it on task-specific data. This section describes the dataset used for fine-tuning the 7 billion parameter version of the Code Llama model as well as the full details of the fine-tuning process.

The dataset for fine-tuning was collected from the APIs-guru OpenAPI definitions directory [62]. The directory contains more than 4,000 definitions in yaml format. Analysis of the repository revealed that about 75% of the definitions in the directory are produced by a handful of major companies like Amazon, Google, and Microsoft. To avoid the dataset bias towards a specific producer, the maximum number of definitions from a single producer was limited to 20. Multiple versions of the same API were also excluded from the dataset as they are likely to contain similar definitions. Definitions from producers used in the evaluation dataset were also excluded from the fine-tuning dataset to avoid data leakage. The resulting dataset contains 990 definitions and can be accessed at the HuggingFace Hub [68].

The authors of the Code Llama model don't reveal the details of the dataset used for training. However, considering its size (500 billion tokens), it would be naive to assume that the publicly available OpenAPI definitions were not used for training. At the same time, the Code Llama model authors claim that fine-tuning the model on already-seen data can still lead to significant performance improvements. This claim is supported by the results of the Python-specialized Code Llama version which was fine-tuned from the base Code Llama model using the subset of the original training dataset.

In order to avoid the objective mismatch between the training and fine-tuning processes, the infilling prompt format was reconstructed to mirror the format described by the Code Llama model authors. Nevertheless, the specifics of the OpenAPI format still had to be taken into account. Similarly to the Code Llama training, the prefix-middle-suffix split was performed at the character level with the split positions selected randomly. The same tokens (<PRE>, <SUF>, <MID>, <EOT>) were used to format the prompt in the PSM and SPM formats and the leading spaces implicitly generated by the SentencePiece tokenizer were suppressed. However, unlike the Code Llama training, the split was performed at the context level and was not limited to documents that fit into the context size as most of the OpenAPI definitions are much larger than the context size used during fine-tuning. In the SPM format, the prefix and middle parts were concatenated after the tokenization to improve the model performance with split tokens which are expected to be encountered often in the real-world OpenAPI completion scenarios.

Another consequence of the large size of the OpenAPI definitions is the bias towards full-context-length completions. As can be seen in Figure 7, the distribution of the training sample lengths is skewed toward the maximum context size of 5,120 tokens when the normal context-level split is used. This bias can cause the model to show optimal performance only when prompted with the same context size as during fine-tuning. To mitigate the bias, the input documents were split into smaller parts of random length between 4 096 and 6 144 tokens (0.8-1.2 of the 5 120 tokens context size) before the context-level split was performed. This technique is further referred to as document splitting. When used before the infilling permutation, document splitting should not be distinguishable from using full documents of uniform length distribution as the truncated prefixes and suffixes correspond to a normal real-world code completion scenario when the context size is limited and doesn't cover the entire document.

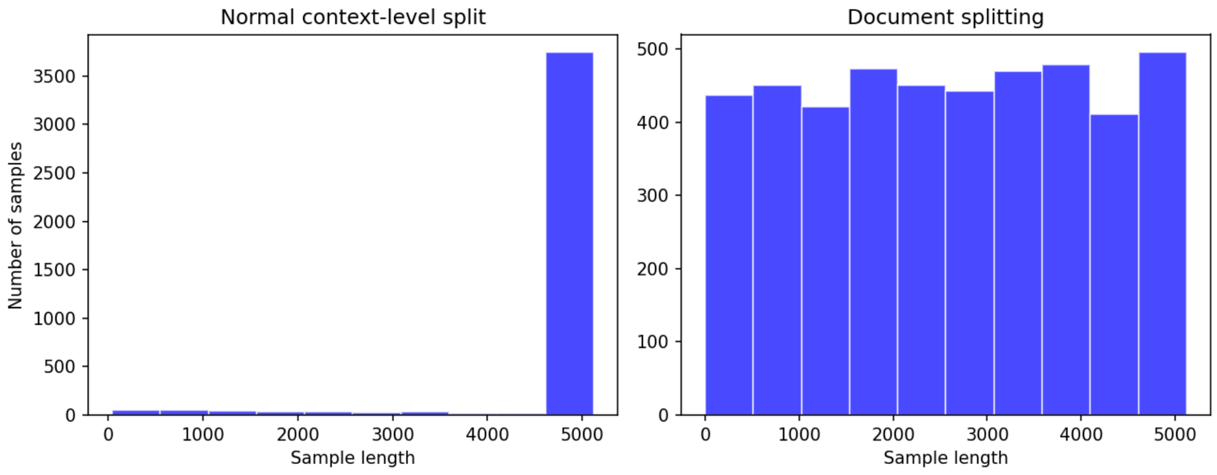


Fig. 7. Sample length distribution with and without document splitting

2.6. Design Conclusion

In this chapter, the methods for the OpenAPI completion solution and benchmark implementation were designed. The key design decisions can be summarized as follows:

1. The criteria for the OpenAPI completion evaluation were defined to be correctness, validity, and speed. The Oasdiff tool was chosen to calculate the semantic difference between the original and the generated OpenAPI definitions and the heuristics were defined to tolerate subjective changes in the definitions.
2. The proposed semantics-aware OpenAPI completion benchmark was designed to resemble the real-world usage of the solution by masking a multiline part of the OpenAPI definition and using semantic analysis to evaluate the completion result. To the best of the author's knowledge, the benchmark is the first publically available evaluation framework for OpenAPI completion.
3. The evaluation dataset was collected from the APIs-guru OpenAPI definitions directory and consists of ten definitions from different domains and API producers. The number of test cases was limited to 100 as commercial solutions like GitHub Copilot require manual evaluation which must be kept in a reasonable time frame.
4. The Code Llama model was selected as the foundational model for the research and the solution implementation. The following criteria were considered when selecting the model: licensing terms, multilanguage support, code infilling capabilities, context size, fine-tuning

support, and community activity.

5. The Hugging Face Inference Endpoints platform was selected as the infrastructure platform for the solution implementation. The platform supports cost-efficient per-request pricing, automated model deployment, simple inference APIs, a wide range of hardware options, automatic scaling and load balancing, and has a big community and good documentation.
6. The Hugging Face Transformers and PEFT libraries were selected for the fine-tuning process. In addition to providing native support for Code Llama model LoRA fine-tuning, the libraries are well-integrated with the Hugging Face Inference Endpoints platform.
7. JavaScript was selected as the primary programming language for the client-side integration layer. The language is compatible with the target platforms (IDE plugin systems and web browsers) and has a developed tooling ecosystem for integration with the Hugging Face Inference Endpoints platform and estimating the Code Llama model token length.
8. The prompt engineering techniques were designed to optimize the performance of the original Code Llama model before proceeding to the fine-tuning process. The prefix-to-suffix ratio, context size, model size, infilling format, and metadata inclusion techniques will be evaluated to find the optimal prompt format for the OpenAPI completion task.
9. The fine-tuning techniques were designed to further optimize the performance of the Code Llama model. A novel document splitting technique was proposed to mitigate the bias towards full-context-length completions that the widely used context-level infilling transformation is prone to. The LoRA PEFT method was selected to optimize the cost-efficiency of the fine-tuning process.
10. The fine-tuning dataset was collected from the APIs-guru OpenAPI definitions directory. To prevent a possible bias towards the design style of major API producers and data leakage to the evaluation benchmark, the dataset was filtered to limit the number of definitions from a single producer to 20 and exclude the definitions that are used in the evaluation dataset.
11. To the best of the author's knowledge, the implemented fine-tuning pipeline is the first publically available solution for fine-tuning the Code Llama model for code infilling. The solution is not limited to OpenAPI completion and can be used to improve the Code Llama model performance in any programming language or domain.

3. OpenAPI Completion Evaluation Results

Following the prompt engineering and fine-tuning methods described in the previous section, the Code Llama model was optimized for the OpenAPI completion task. This section presents the results of the evaluation of the optimized model and compares it to the performance of GitHub Copilot. For all the evaluations described in this section, the same set of masked OpenAPI definitions was used ensuring consistent evaluation conditions.

3.1. GitHub Copilot Performance in OpenAPI Completion

GitHub Copilot was selected as the baseline solution for OpenAPI generation as at the time of writing it is the most widely used coding assistant on the market. The solution is a commercial product available with a monthly subscription plan.

As GitHub Copilot does not provide an API for programmatic access, the completion cannot be automated and must be performed manually. For the evaluation results to be comparable with the automatic completion results, the manual evaluation process resembles the automatic completion procedure as closely as possible and uses the same set of masked OpenAPI definitions. Manual completion was performed for each masked definition following these steps:

1. The «MASK» marker is removed, and the cursor is set in place of the marker.
2. If Copilot refuses to generate a suggestion, the last character is removed and added again to trigger the completion.
3. Copilot's suggestions are accepted without modifications one by one until one of the following conditions is reached:
   (a) Suggestions are no longer generated (Copilot considers the infilling complete).
   (b) Maximum number of completion lines is reached (15 lines).

The speed of suggestion generation using GitHub Copilot was measured by tracking the time between triggering completion with a key press and seeing the generated suggestion on the screen. The number of characters generated was divided by the generation time. The experiment was repeated ten times to take human error as well as random service latency factors into account. The average code generation speed was measured to be 19.3 characters per second with a standard deviation of 1.9.

Following the completion procedure described, GitHub Copilot correctly completed 29% of the OpenAPI definitions. 68% of the completed definitions were valid.

3.2. Original Code Llama Model for OpenAPI Completion

Using the benchmark proposed in the previous chapter, an extensive set of experiments was performed to analyze the impact of a variety of variables and prompt engineering techniques on the performance of the Code Llama model in the OpenAPI completion task. The experiments were designed to analyze the impact of a single variable at a time. When not stated otherwise, the 7 billion parameter version of the Code Llama model, the PSM prompt format (1), the 50/50 prefix-to-suffix ratio, and the 4 096 tokens context size were used.

The Code Llama models were evaluated at the Hugging Face Inference Endpoints platform [66] using the configuration described in Table 3.

Table 3. Hugging Face Inference Endpoints configuration used for Code Llama models evaluation

Model	Configuration Option	Value
Code Llama 7B	Instance Type	Nvidia A10G · 1x GPU · 24 GB
	Task	Text Generation
	Model Revision	bc52832
	Custom Cuda Kernels	Enabled
	Quantization	None
	Max Input Length	8 096
	Max Number of Tokens	9 120
	Max Batch Prefill Tokens	8 096
	Max Batch Total Tokens	9 120
Code Llama 13B	Instance Type	Nvidia A10G · 4x GPU · 96 GB
	Task	Text Generation
	Model Revision	a49a368
	Custom Cuda Kernels	Enabled
	Quantization	None
	Max Input Length	4 096
	Max Number of Tokens	4 352
	Max Batch Prefill Tokens	4 096
	Max Batch Total Tokens	4 352

3.2.1. Optimal Prefix-to-suffix Ratio

A range of prefix-to-suffix ratios from 30/70 up to 90/10 was evaluated to find the optimal proportion. The evaluation was performed on the 7 billion parameter version of the Code Llama model with a constant context size of 4 096 tokens. The optimal prefix-to-suffix ratio can depend on the mentioned factors but for the purpose of parameter space reduction, the dependencies are intentionally disregarded in this research.

Figure 8 depicts the dependency of correctness and validity of the OpenAPI completion on the prefix-to-suffix ratio. As can be seen from the graph, the highest percentage of correctly infilled definitions of 36% corresponds to 50/50 and 60/40 prefix-to-suffix ratios. The 50/50 ratio demonstrates a higher validity percentage of 63% compared to 61% for the 60/40 ratio. Even though the dip at the 60/40 ratio can be attributed to the factor of randomness, the 50/50 ratio can still be considered optimal as it does not discriminate the importance of either prefix or suffix without sound justification.

3.2.2. Optimal Context Size

Context size is another factor that affects not only the solution performance but also the speed of code generation and, consequently, throughput and resource efficiency of a real-world implementation. A range of context sizes from 1 024 to 7 168 tokens was evaluated with a constant prefix-to-suffix ratio of 50/50 and the 7 billion parameter version of the Code Llama model.

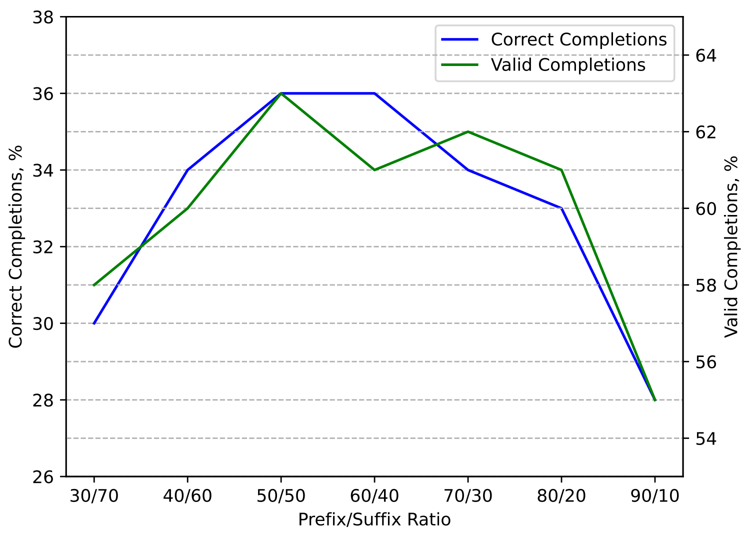


Fig. 8. OpenAPI completion performance of the original Code Llama 7B model at a range of prefix-to-suffix ratios with a fixed context size of 4096 tokens

As can be seen in Figure 9, OpenAPI completion correctness and validity increase with the expanding context size. The correctness rate, which is the primary target metric in this research, demonstrates a much smoother trend with a significant increase up to the context size of 4 096 tokens. After this point, the correctness level stabilizes at around 36% and even decreases by 1% after 6 144 tokens. The speed, as expected, follows the downward trend with a dramatic decline after the 3 072 tokens point. This can be explained by the quadratic complexity of the input processing in decoder-only transformers.

With a correctness rate of 36%, validity rate of 63%, and a generation speed of 79 characters per second, the context size of 4 096 tokens can be considered optimal for real-world scenarios since after this point, neglectable correctness and validity increases come at a cost of significant speed and resource-efficiency deterioration.

3.2.3. Autoregressive Generation and Infilling Modes

This experiment evaluated the Code Llama model in the traditional autoregressive code generation mode with a range of context sizes from 1 024 to 7 168 tokens. As expected, the performance in this mode is inferior to the infilling mode with 50/50 prefix-to-suffix ratio in terms of both the correctness and validity of the generated OpenAPI completions (Figure 10). For example, the model correctly completed only 10% of the definitions and generated valid completions 24% of the time, which is a 72% correctness decrease compared to the same configuration in the infilling mode.

The results of this experiment demonstrate that the infilling mode is essential for code completion tasks regardless of the amount of context provided to the model.

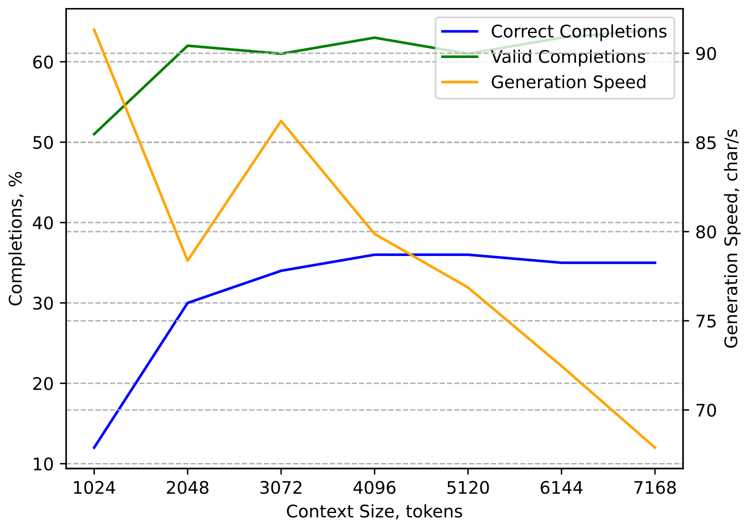


Fig. 9. OpenAPI completion performance of the original Code Llama 7B model at a range of context sizes with a fixed prefix-to-suffix-ratio of 50/50

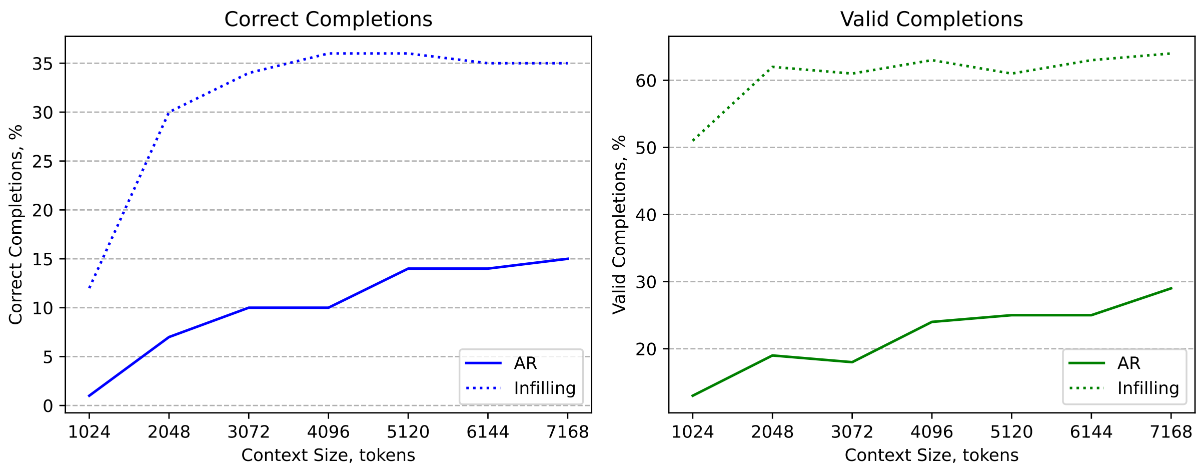


Fig. 10. OpenAPI completion performance of the original Code Llama 7B model in autoregressive generation and infilling modes

3.2.4. PSM and SPM Infilling Formats

SPM format (2) is an alternative prompt format supported by the Code Llama model in the infilling mode. Contrary to the fact that the SPM format demonstrated higher performance in the original work of Bavarian et al. [28], the authors of Code Llama designed PSM (1) to be the primary prompting format for the model stating that the SPM implementation might require additional token healing on the tokenization step at the inference time. In this experiment, the default Code Llama tokenizer is used. As can be seen in Figure 11, the SPM prompting format underperforms at all context sizes compared to PSM. For example, the model correctly completed 24% of the definitions and generated valid completions 53% of the time at 4 096 tokens, which is a 33.3% correctness decrease compared to the same configuration with the PSM prompt format.

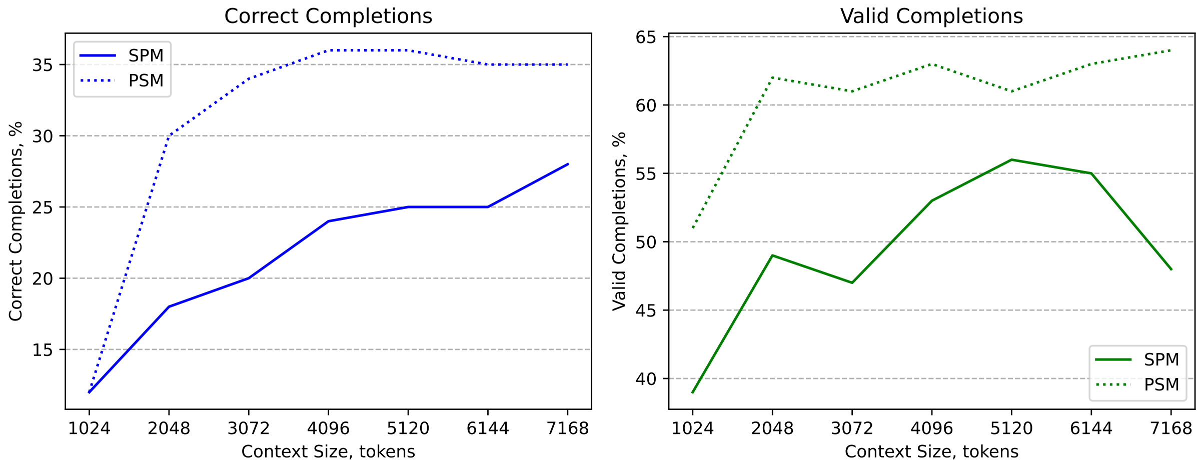


Fig. 11. OpenAPI completion performance of the original Code Llama 7B model with the PSM and SPM prompt formats

3.2.5. OpenAPI Metadata in Prompt

The experiment was performed on the 7 billion parameter Code Llama model with a 60/40 prefix-to-suffix ratio and a range of context sizes from 4 096 to 7 168 tokens. The extended prefix size was expected to compensate for the space taken by the components section in the prompt.

As can be seen in Figure 12, adding the components list to the prompt prefix results in moderate validness and correctness improvements at the larger context sizes (11.4% correctness and 1.6% validness improvement at 6 144 tokens). The results at a smaller context size of 4 096 tokens, on the other hand, demonstrate a significant drop in both correctness (25%) and validity (6.3%).

The obtained results might mean that the components are not usually required for correct OpenAPI completion. The model can deduct the correct component names from just the surrounding context. When used with a relatively small context size, the suggested method can even be harmful to the solution performance as it consumes space that could otherwise be taken by more valuable information in the prefix.

3.2.6. Optimal Model Size

The Code Llama model is available in three sizes: 7 billion parameters, 13 billion parameters, and 34 billion parameters. The impact of the model size on the solution performance is evaluated by comparing the performance of the 7 billion parameter model with the performance of the 13 billion parameter model. Due to resource constraints and the inability of the 34 billion parameter model to work in the infilling mode, this model size is not evaluated in the experiment.

A machine with four Nvidia A10G GPUs and a total of 96 GB of GPU memory was used for the evaluation. This machine made it possible to evaluate the model with up to 4 096 tokens of context. The evaluation was performed on a range of context sizes from 1 024 to 4 096 tokens with a constant prefix-to-suffix ratio of 50/50.

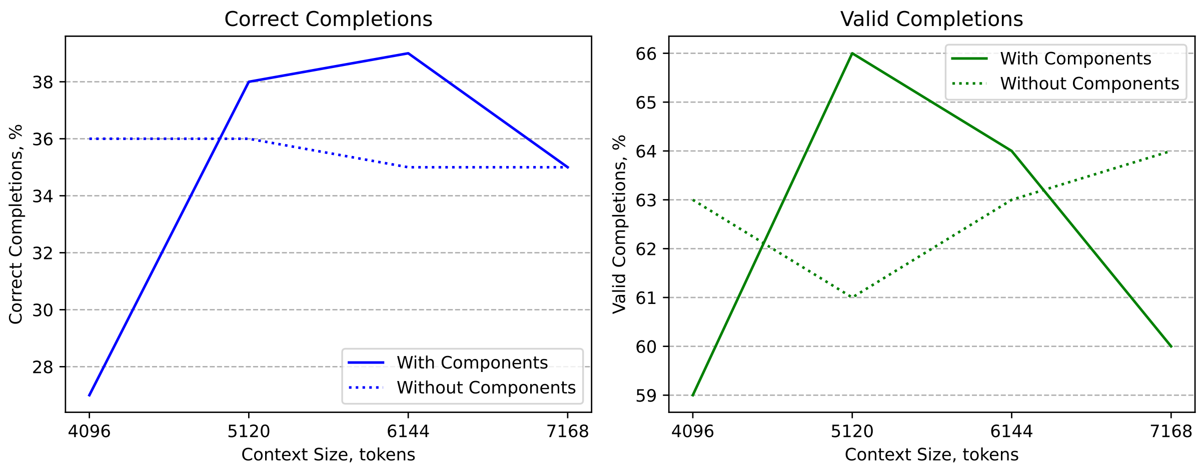


Fig. 12. OpenAPI completion performance of the original Code Llama 7B model with and without OpenAPI components in the prompt prefix

As can be seen in Figure 13, the bigger 13 billion parameter model outperforms the smaller model at all context sizes with the only exception being correctness after 3 072 tokens. At the context size of 4 096 tokens, Code Llama 13B surpasses the results obtained with Code Llama 7B by only 1% in validity while falling behind in correctness by 5%.

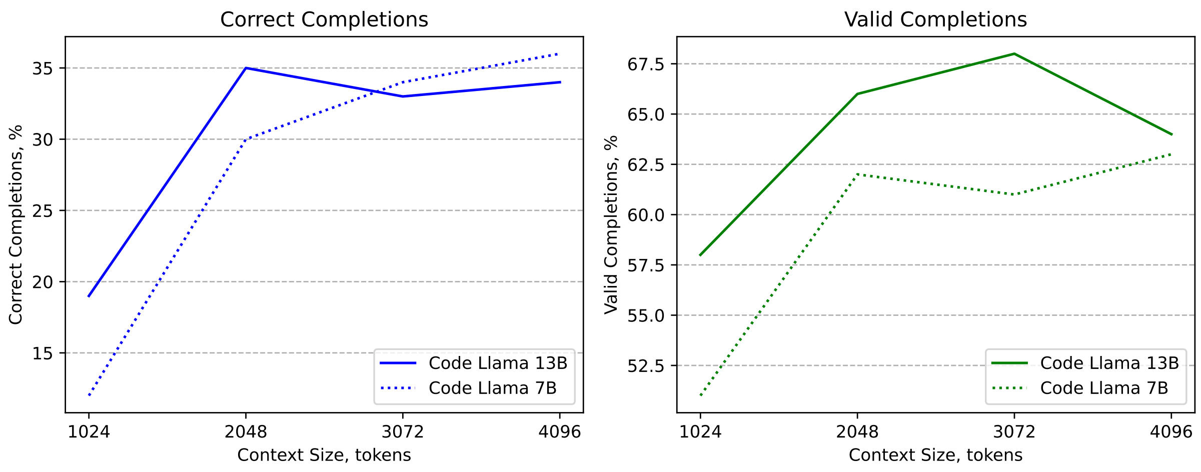


Fig. 13. OpenAPI completion performance of the 7 and 13 billion parameter versions of the original Code Llama model

Figure 14 compares the generation speed trends of the models. Due to the GPU computing power redundancy caused by the high memory requirements of the model, the 13 billion Code Llama outperforms the smaller model at 1 024 and 2 048 tokens. However, the quadratic complexity of the attention mechanism quickly makes up for the computing power redundancy, and the generation speed drops below the speed of the smaller model at 3 072 and 4 096 tokens. The speed reduction trend suggests that the difference in the generation speed will only increase with the context size.

3.2.7. YAML and JSON OpenAPI Fromats

The OpenAPI specification allows using both YAML and JSON formats for the definitions. The Code Llama model operates on the character level and does not distinguish between the two

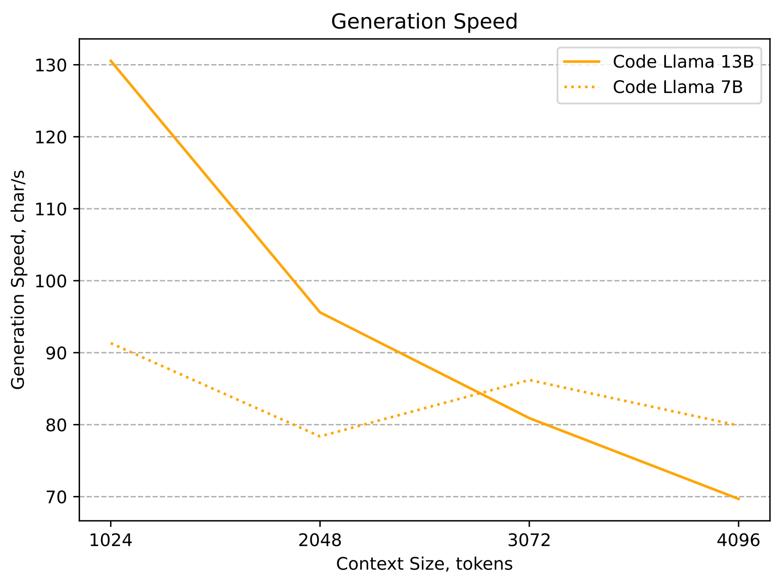


Fig. 14. Generation speed of the 7 and 13 billion parameter versions of the original Code Llama model

formats. However, the performance in these formats is expected to be different due to different levels of syntactic complexity. As can be seen in Figure 15, two semantically identical OpenAPI snippets in YAML and JSON can require a significantly different number of characters and tokens to be represented. In this case, JSON takes 43% more characters and 51% more tokens than YAML when the Llama 2 tokenizer is used. As discussed in the previous chapter, this tokenizer can be considered identical to the Code Llama tokenizer for this experiment.

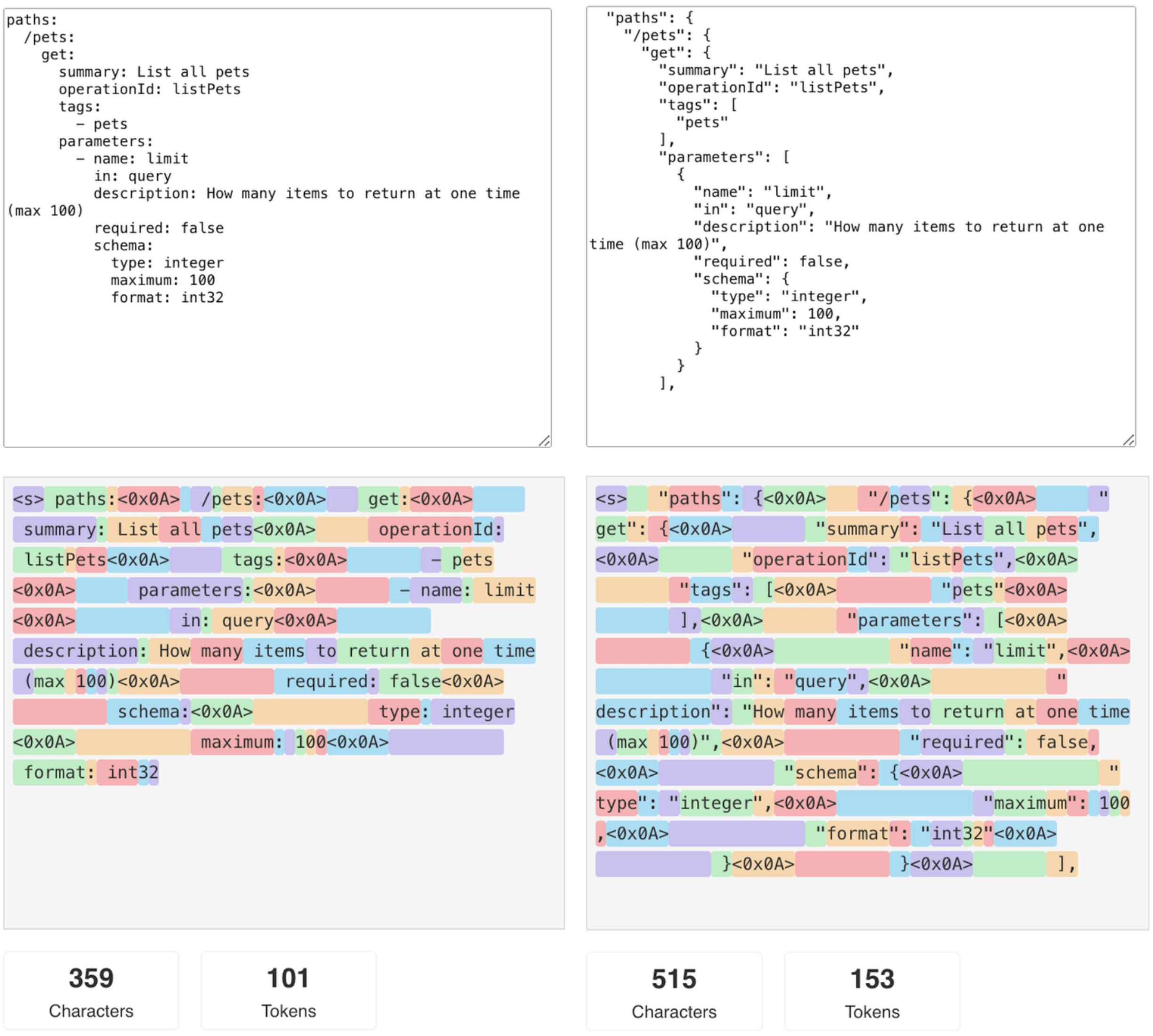
Infilling the same set of OpenAPI definitions converted to JSON using Code Llama 7B model with a 50/50 prefix-to-suffix ratio and 4 096 tokens of context resulted in 27% of correctly infilled definitions which is 25% less than the performance of the same model on the original YAML definitions. The validity of the generated definitions was 56% which is 11.1% less than the validity of the YAML definitions. The generation speed was 72 characters per second which is 8% less than the speed of the same model on the original YAML definitions. The results suggest that the JSON format is more difficult for the model to generate.

3.3. Fine-tuning Code Llama Model for OpenAPI Completion

As demonstrated in the previous section, the original Code Llama model is already capable of OpenAPI completion and even outperforms GitHub Copilot. This section evaluates the fine-tuning approaches that can further improve the model performance.

3.3.1. Benefits of Using a Mixture of PSM and SPM Infilling Formats

The study of Bavarian et al. suggests that using a mix of SPM and PSM formats during training leads to an increase in the infilling performance [28]. At the same time, the original Code Llama model has a known problem with infilling in the SPM format 3.2.4. It is worth evaluating the efficiency of including this format in the fine-tuning process. This section compares the performance of the 7 billion parameter Code Llama model fine-tuned solely with the PSM format and with a mixture of PSM and SPM formats (joint fine-tuning). When used, the SPM format was applied to half of the training samples. Context size of 4 096 tokens was used during



```
paths:
  /pets:
    get:
      summary: List all pets
      operationId: listPets
      tags:
        - pets
      parameters:
        - name: limit
          in: query
          description: How many items to return at one time
(max 100)
          required: false
          schema:
            type: integer
            maximum: 100
            format: int32
```

```
  "paths": {
    "/pets": {
      "get": {
        "summary": "List all pets",
        "operationId": "listPets",
        "tags": [
          "pets"
        ],
        "parameters": [
          {
            "name": "limit",
            "in": "query",
            "description": "How many items to return at one
time (max 100)",
            "required": false,
            "schema": {
              "type": "integer",
              "maximum": 100,
              "format": "int32"
            }
          }
        ],
```

```
<s> paths:<0x0A>  /pets:<0x0A>    get:<0x0A>
 summary: List all pets<0x0A>      operationId:
 listPets<0x0A>      tags:<0x0A>        - pets
<0x0A>      parameters:<0x0A>        - name: limit
<0x0A>          in: query<0x0A>
 description: How many items to return at one time
 (max 100)<0x0A>          required: false<0x0A>
          schema:<0x0A>            type: integer
<0x0A>            maximum: 100<0x0A>
 format: int32
```

```
<s>   "paths": {<0x0A>    "/pets": {<0x0A>      "
get": {<0x0A>        "summary": "List all pets",
<0x0A>        "operationId": "listPets",<0x0A>
        "tags": [<0x0A>          "pets"<0x0A>
        ],<0x0A>        "parameters": [<0x0A>
          {<0x0A>            "name": "limit",<0x0A>
            "in": "query",<0x0A>            "
description": "How many items to return at one time
 (max 100)",<0x0A>            "required": false,
<0x0A>            "schema": {<0x0A>              "
type": "integer",<0x0A>              "maximum": 100
,<0x0A>              "format": "int32"<0x0A>
            }<0x0A>          }<0x0A>        ],
```

359 Characters | 101 Tokens | 515 Characters | 153 Tokens

Fig. 15. YAML and JSON tokenization comparison

fine-tuning.

As can be seen in Figure 16, both fine-tuning approaches demonstrate similar performance in the PSM format with the joint fine-tuning showing a slight improvement in correctness and validity. The joint fine-tuning also results in significantly higher performance when the SPM format is used for inference. These results suggest that it always makes sense to include the SPM format when fine-tuning the Code Llama model for code infilling tasks.

Compared to the original Code Llama model, the model fine-tuned with the joint format demonstrates a peak improvement of 28.6% in both correctness and validity at the context size of 6 144 tokens when prompted with the PSM format. Performance in the SPM format is still inferior to PSM but the gap is reduced from 28.6% to 11.1% in correctness and from 12.7% to 4.9% in validity at the same context size. This shows that the selected fine-tuning approach can also lead to solving the original model's problem of split-token handling in the SPM format.

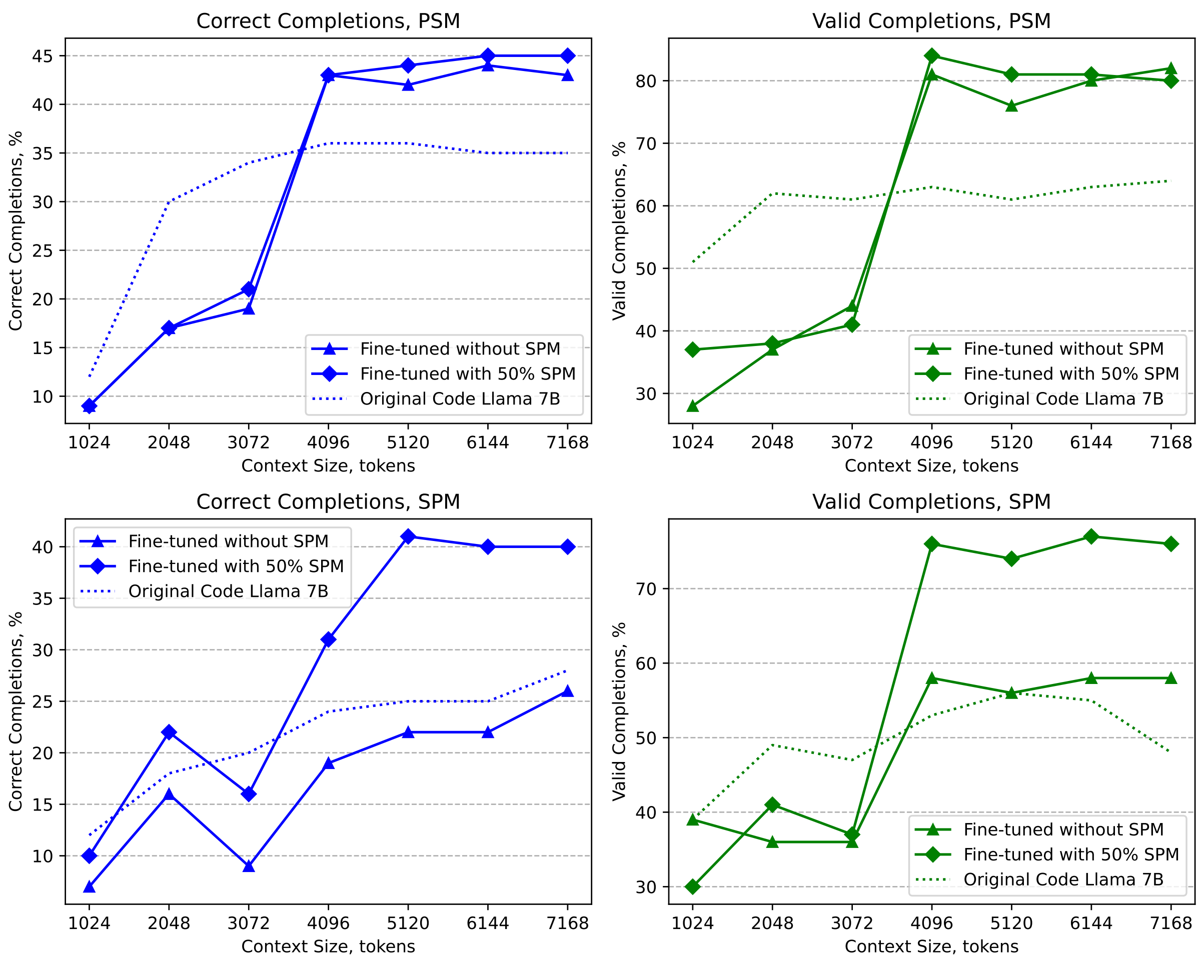


Fig. 16. OpenAPI completion performance of the Code Llama 7B model fine-tuned solely with the PSM format and with a mixture of PSM and SPM formats

3.3.2. Full-context-length Infilling Bias and Document Splitting Technique

As revealed in the previous section, the fine-tuned model underperforms when prompted with context sizes smaller than used during fine-tuning. This can be caused by the bias towards full-context-length completions discussed in Section 2.5. This section further evaluates the impact of the fine-tuning context size as well as the suggested document splitting technique on the performance of the fine-tuned model.

As can be seen in Figure 17, both 4 096 tokens and 5 120 tokens fine-tuned models underperform compared to the original Code Llama model at context sizes smaller than the corresponding fine-tuning context sizes. After this point, the performance of the fine-tuned models stabilizes. Detailed inspection of the infilling results shows that the fine-tuned models tend to generate completions that are too long and contain irrelevant information when prompted with smaller context sizes.

A tempting idea to mitigate the problem is to use a much smaller context size for fine-tuning. As demonstrated in Figure 17, using the context size of 2 048 tokens for fine-tuning indeed decreases the performance decline at smaller context sizes. However, using smaller context also reduces the performance gains at larger context sizes to the point where the fine-tuned model underperforms compared to the original Code Llama model. Nevertheless, the model still demonstrates the best validity rates at all context sizes.

The model fine-tuned with the document splitting technique demonstrates more equalized performance across the context sizes. The correctness and validity of the completions generated by this model surpass the original Code Llama model at almost all context sizes with a peak increase of 20% at 6 144 and 7 168 tokens and a marginal decrease of 5.8% at 3 072 tokens. The model slightly underperforms compared to the model fine-tuned with the normal context-level split technique at the context size of 4 096 tokens when prompted with context sizes larger than 4 096 tokens, but it outperforms the same model at smaller context sizes by a significant margin.

The loss measured during the training process also appears to be more stable when the document splitting technique is applied (Appendix 1). This can be explained by a more granular shuffling of the smaller training samples which can lead to a more stable training process.

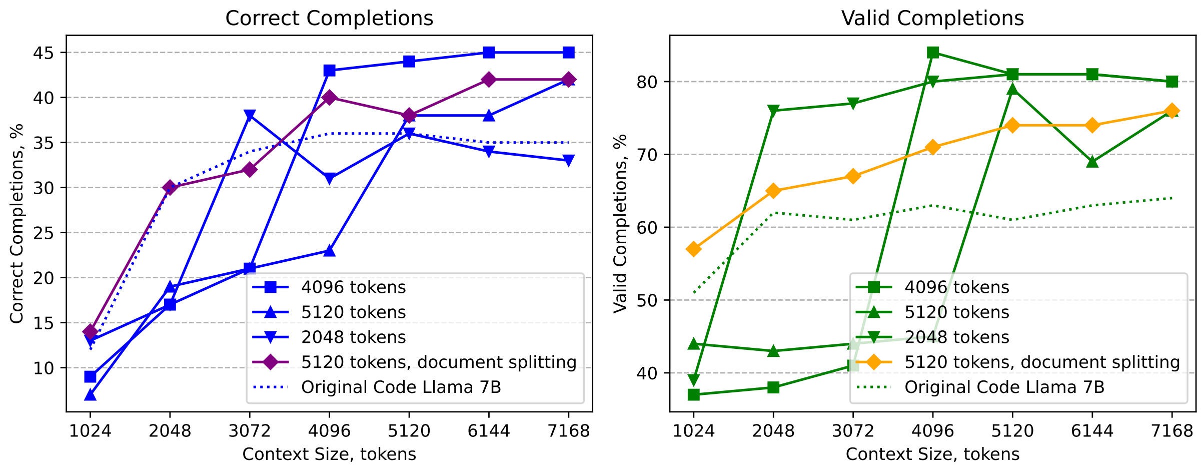


Fig. 17. OpenAPI completion performance of the Code Llama 7B model fine-tuned at different context sizes and with the document splitting technique

3.4. Experiments Conclusion

Through the course of this research, the impact of multiple variables, prompt engineering, and fine-tuning techniques on the OpenAPI completion performance of the Code Llama model was evaluated. The most notable experimental results are summarized and compared with one of the most widely adopted commercial solutions – GitHub Copilot in Table 4.

1. Original Code Llama model is capable of outperforming GitHub Copilot. The Code Llama 7B model, when used with the optimal prefix-to-suffix ratio of 50/50, the PSM prompt format, and the context size of 4 096 tokens, outperforms GitHub Copilot by 24.1% in correctness, generating correct OpenAPI completions 36% of the time. At the same time, the model produces valid completions 63% of the time which is a 7.3% decrease compared to Copilot. It's worth pointing out that the GPT-3 model, which is the foundational model used by GitHub Copilot, uses 175 billion parameters which is 25 times more than in the case of the Code Llama model.

Table 4. OpenAPI completion performance comparison

Model / Solution	Correctness, max., %	Validness, max., %	Correctness, avg., %	Validity, avg., %
GitHub Copilot	29.0	68.0	29.0	68.0
Code Llama 7B	36.0 (4 096 tokens)	64.0 (7 168 tokens)	31.1	60.7
Code Llama 13B (evaluated at 1 024-4 096 tokens)	34.0 (4 096 tokens)	68.0 (3 072 tokens)	30.2	64.0
Code Llama 7B, autoregressive mode	15.0 (7 168 tokens)	29.0 (7 168 tokens)	10.1	21.9
Code Llama 7B, fine-tuned at 4 096 tokens	**45.0** (6 144 tokens)	**84.0** (4 096 tokens)	32.0	63.1
Code Llama 7B, fine-tuned at 5 120 tokens	42.0 (7 168 tokens)	79.0 (5 120 tokens)	26.8	57.1
Code Llama 7B, fine-tuned with document splitting	42.0 (6 144 tokens)	76.0 (7 168 tokens)	**34.0**	**69.1**

2. Larger model does not always show significant performance improvement. The larger Code Llama 13B model demonstrated only a marginal correctness improvement compared to the smaller 7 billion parameter version. This might mean that the OpenAPI format is not complex enough to leverage the additional model capacity. Considering the quadrupled computational cost, this draws the larger versions of the model (including 13, 34, and 70 billion parameter versions) not practical for use in OpenAPI completion.
3. Infilling capabilities are essential for the code completion task. The Code Llama 7B model, prompted in autoregressive mode severely underperforms compared to the same experiment performed in the infilling mode. The correctness decrease of 72.2% and the validity decrease of 61.9% at 4 096 tokens context size served as another argument for not evaluating the bigger 34 billion version of the model which was not trained for code infilling (see Section 2.2.1).
4. The YAML format is not only easier to work with for humans but also for the Code Llama model. The model demonstrated a 25% correctness decrease and an 11.1% validity decrease when prompted with the same set of OpenAPI definitions converted to JSON. The tokenization results of the YAML and JSON formats suggest that the JSON format is more complex for the model for the same reason it is more complicated for humans – the ratio of the syntactic characters to the semantic information is higher in JSON.
5. Task-specific fine-tuning can significantly improve the model performance. The 7 billion parameter version of the Code Llama model fine-tuned for OpenAPI completion at the context size of 4 096 tokens following the methods described in Section 2.5 demonstrates a maximum correctness improvement of 28.6% compared to the original model and further outperforms GitHub Copilot by 55.2%.

6. Normal context-level infilling transformation as used in the original Code Llama model training described in Section 2.2.1 can lead to suboptimal performance at smaller context sizes. The case of OpenAPI completion revealed that, when used with documents that are significantly larger than the fine-tuning context size, this technique leads to a bias towards full-context-length completions. This bias can cause the model to generate too long and irrelevant completions when prompted with a context size smaller than the fine-tuning context size. The models trained with this technique (Code Llama 7B, fine-tuned at 4 096 tokens and Code Llama 7B, fine-tuned at 5 120 tokens) still demonstrate the best performance at the context size used during fine-tuning which can be leveraged when the inference context size is known in advance.
7. The document splitting technique proposed in Section 2.5 can be used to mitigate the problem of the full-context-length infilling bias and lead to more unified performance across the context sizes. Even considering the fact that the models fine-tuned without document splitting outperform the model fine-tuned with this technique at the context size used during fine-tuning, the latter demonstrates the best average performance across the context sizes which is crucial for training foundational or open-source models which are expected to be used with a variety of context sizes.

Conclusion

The primary aim of investigating the approaches for developing an OpenAPI completion solution that would outperform the current state-of-the-art solutions in this task can be considered accomplished. The solution proposed in this research is based on an open-source LLM from Meta – Code Llama. As a result of a series of experiments, the optimized model demonstrates a peak correctness improvement of 55.2% compared to one of the most widely used commercial code completion solutions – GitHub Copilot. Based on the objectives of the research, the following conclusions can be made:

1. The analysis of the current state of the art in code generation and evaluation revealed the following challenges and opportunities for the research: (1) modern code completion solutions are based on LLMs which require large amounts of data and computational resources to train; (2) code generation LLMs are usually trained on an unbound or a large set of programming languages and tasks which means that OpenAPI-specific optimization can lead to a competitive advantage in this area; (3) the existing code generation evaluation methods are limited to a subset of popular programming languages and an OpenAPI completion benchmark has not been developed yet.
2. GitHub Copilot, one of the most widely adopted commercial code generation and completion solutions was selected as the baseline solution for the study. Code Llama, an open-source LLM from Meta was selected as the foundational model for the proposed solution implementation due to its permissive license, best-in-class performance, large context size support, and infilling capabilities.
3. Three metrics were proposed to evaluate the performance of the OpenAPI completion solutions: correctness, validity, and generation speed. The correctness metric aims to closely represent the experience of a real-world user by tolerating the possible variations from the ground truth completion. The validity metric evaluates the rate of syntactically correct completions. The generation speed metric is only used to evaluate the trends in the performance of the solutions as the generation speed depends on the hardware which was not always under control during the experiments. To the best of the author's knowledge, the proposed semantics-aware OpenAPI completion benchmark is the first publically available evaluation framework for OpenAPI completion. The benchmark's source code is made available on GitHub [1].
4. The OpenAPI completion solution proposed in this research is based on a modular architecture that enables easy alteration of the prompt building, infilling, and result embedding strategies. The solution was implemented and published as a JavaScript package that can be used to add intelligent OpenAPI completion capabilities to any text editor or IDE that supports plugins in JavaScript as well as to any web-based code editor. The solution requires an LLM to be deployed on the Hugging Face Inference Endpoints platform and is not limited to the Code Llama model. The source code is made available on GitHub [3] and the JavaScript package is made available on NPM [69].
5. The following prompt engineering and fine-tuning techniques were proposed based on the analysis of the foundational model – Code Llama: (1) optimal prefix-to-suffix

ratio search, (2) optimal context size search, (3) optimal model size search, (4) optimal generation mode and prompt format search, (5) inclusion of OpenAPI metadata in the prompt, (6) fine-tuning with a mixture of PSM and SPM infilling formats, (7) fine-tuning with the novel document splitting technique. The document splitting technique proposed in this research was demonstrated to be effective in mitigating the full-context-length infilling bias, a common problem in the widely used fill-in-the-middle training objective for code generation LLMs even outside the OpenAPI completion task. To the best of the author's knowledge, this is the first study describing Code Llama fine-tuning in detail. The fine-tuning pipeline source code is made available on GitHub [2].

6. The following key findings were obtained from the experiments conducted in this research: (1) the original Code Llama model is already capable of outperforming GitHub Copilot in OpenAPI completion; (2) the larger model does not always show significant performance improvement; (4) infilling capabilities are essential for the code completion task; (5) the YAML format is not only easier to work with for humans but also for the Code Llama model; (6) task-specific fine-tuning can significantly improve the model performance; (7) normal context-level infilling transformation as used in the original Code Llama model can lead to suboptimal performance at smaller context sizes; (8) the document splitting technique proposed in this study leads to a more unified performance across context sizes. A more detailed analysis of the experimental results and their implications is provided in Section 3.4.

At the time of writing, part of this research is being submitted for publication in a peer-reviewed scientific journal. The preprint of the article is available on Arxiv [4].

List of references

1. *GitHub - BohdanPetryshyn/openapi-completion-benchmark: OpenAPI completion benchmark for the paper "Optimizing Large Language Models for OpenAPI Completion: A Comparative Analysis of Code Llama and GitHub Copilot" — github.com* [`https://github.com/BohdanPetryshyn/openapi-completion-benchmark`]. [S. a.]. [Accessed 27-05-2024].
2. *GitHub - BohdanPetryshyn/code-llama-fim-fine-tuning — github.com* [`https://github.com/BohdanPetryshyn/code-llama-fim-fine-tuning`]. [S. a.]. [Accessed 27-05-2024].
3. *GitHub - BohdanPetryshyn/hf-code-llama-infiller: Code infilling with Hugging Face Code Llama — github.com* [`https://github.com/BohdanPetryshyn/hf-code-llama-infiller`]. [S. a.]. [Accessed 27-05-2024].
4. PETRYSHYN, Bohdan; Mantas LUKOŠEVIČIUS. *Optimizing Large Language Models for OpenAPI Code Completion*. 2024. Pasiekiamas per arXiv: `2405.15729 [cs.SE]`.
5. DEHAERNE, Enrique ir kt. Code generation using machine learning: A systematic review. *Ieee Access*. 2022, t. 10, p. 82434–82455.
6. VASWANI, Ashish ir kt. Attention is all you need. *Advances in neural information processing systems*. 2017, t. 30.
7. BAHDANAU, Dzmitry; Kyunghyun CHO; Yoshua BENGIO. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*. 2014.
8. SALEHINEJAD, Hojjat ir kt. Recent advances in recurrent neural networks. *arXiv preprint arXiv:1801.01078*. 2017.
9. GLOROT, Xavier; Yoshua BENGIO. Understanding the difficulty of training deep feedforward neural networks. *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop ir Conference Proceedings, 2010, p. 249–256.
10. DEVLIN, Jacob; Ming-Wei CHANG; Kenton LEE; Kristina TOUTANOVA. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*. 2018.
11. LIU, Yinhan ir kt. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*. 2019.
12. LAN, Zhenzhong ir kt. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*. 2019.
13. HE, Pengcheng; Xiaodong LIU; Jianfeng GAO; Weizhu CHEN. Deberta: Decoding-enhanced bert with disentangled attention. *arXiv preprint arXiv:2006.03654*. 2020.
14. RAFFEL, Colin ir kt. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*. 2020, t. 21, Nr. 140, p. 1–67.
15. LEWIS, Mike ir kt. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*. 2019.
16. RADFORD, Alec; Karthik NARASIMHAN; Tim SALIMANS; Ilya SUTSKEVER ir kt. Improving language understanding by generative pre-training. 2018.

17. CHOWDHERY, Aakanksha ir kt. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*. 2023, t. 24, Nr. 240, p. 1–113.
18. HOFFMANN, Jordan ir kt. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*. 2022.
19. TOUVRON, Hugo ir kt. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*. 2023.
20. TOUVRON, Hugo ir kt. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*. 2023.
21. WANG, Thomas ir kt. What language model architecture and pretraining objective works best for zero-shot generalization? : *International Conference on Machine Learning*. PMLR, 2022, p. 22964–22984.
22. REYNOLDS, Laria; Kyle MCDONELL. Prompt programming for large language models: Beyond the few-shot paradigm. *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*. 2021, p. 1–7.
23. LI, Yujia ir kt. Competition-level code generation with alphacode. *Science*. 2022, t. 378, Nr. 6624, p. 1092–1097.
24. CHEN, Mark ir kt. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*. 2021.
25. NIJKAMP, Erik ir kt. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*. 2022.
26. NIJKAMP, Erik ir kt. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309*. 2023.
27. ROZIERE, Baptiste ir kt. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*. 2023.
28. BAVARIAN, Mohammad ir kt. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*. 2022.
29. GAO, Andrew. Prompt engineering for large language models. *Available at SSRN 4504303*. 2023.
30. WEI, Jason ir kt. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*. 2022, t. 35, p. 24824–24837.
31. LIU, Jiacheng ir kt. Generated knowledge prompting for commonsense reasoning. *arXiv preprint arXiv:2110.08387*. 2021.
32. DING, Ning ir kt. Parameter-efficient fine-tuning of large-scale pre-trained language models. *Nature Machine Intelligence*. 2023, t. 5, Nr. 3, p. 220–235.
33. HOULSBY, Neil ir kt. Parameter-efficient transfer learning for NLP. *International conference on machine learning*. PMLR, 2019, p. 2790–2799.
34. ZAKEN, Elad Ben; Shauli RAVFOGEL; Yoav GOLDBERG. Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language-models. *arXiv preprint arXiv:2106.10199*. 2021.
35. HU, Edward J ir kt. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*. 2021.
36. *GitHub Copilot · Your AI pair programmer — github.com* [`https://github.com/features/copilot`]. [S. a.]. [Accessed 24-03-2024].

37. *copilot-explorer — thakkarparth007.github.io* [https://thakkarparth007.github.io/copilot-explorer/posts/copilot-internals.html]. [S. a.]. [Accessed 24-03-2024].
38. NGUYEN, Nhan; Sarah NADI. An empirical evaluation of GitHub copilot's code suggestions. *Proceedings of the 19th International Conference on Mining Software Repositories*. 2022, p. 1–5.
39. SOBANIA, Dominik; Martin BRIESCH; Franz ROTHLAUF. Choose your programming copilot: a comparison of the program synthesis performance of github copilot and genetic programming. *Proceedings of the genetic and evolutionary computation conference*. 2022, p. 1019–1027.
40. DAKHEL, Arghavan Moradi ir kt. Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software*. 2023, t. 203, p. 111734.
41. WERMELINGER, Michel. Using GitHub Copilot to Solve Simple Programming Problems.(2023). 2023.
42. *Tabnine is an AI assistant that speeds up delivery and keeps your code safe | Tabnine — tabnine.com* [https://www.tabnine.com/]. [S. a.]. [Accessed 24-03-2024].
43. *AI Code Generator - Amazon CodeWhisperer - AWS — aws.amazon.com* [https://aws.amazon.com/codewhisperer/]. [S. a.]. [Accessed 24-03-2024].
44. *Cody | AI coding assistant — sourcegraph.com* [https://sourcegraph.com/cody]. [S. a.]. [Accessed 24-03-2024].
45. *GitHub - continuedev/continue: The easiest way to code with any LLM—Continue is an open-source autopilot for VS Code and JetBrains — github.com* [https://github.com/continuedev/continue]. [S. a.]. [Accessed 24-03-2024].
46. LUO, Ziyang ir kt. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*. 2023.
47. *Phind Model* [www.phind.com/blog/phind-model-beats-gpt4-fast]. [S. a.]. [Accessed 24-03-2024].
48. JIANG, Albert Q ir kt. Mistral 7B. *arXiv preprint arXiv:2310.06825*. 2023.
49. *GitHub - juyongjiang/CodeUp: CodeUp: A Multilingual Code Generation Llama2 Model with Parameter-Efficient Instruction-Tuning on a Single RTX 3090 — github.com* [https://github.com/juyongjiang/CodeUp]. [S. a.]. [Accessed 24-03-2024].
50. TUNSTALL, Lewis ir kt. Zephyr: Direct distillation of lm alignment. *arXiv preprint arXiv:2310.16944*. 2023.
51. ANIL, Rohan ir kt. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*. 2023.
52. *GitHub - huggingface/llm-vscode: LLM powered development for VSCode — github.com* [https://github.com/huggingface/llm-vscode]. [S. a.]. [Accessed 24-03-2024].
53. SVYATKOVSKIY, Alexey; Shao Kun DENG; Shengyu FU; Neel SUNDARESAN. Intellicode compose: Code generation using transformer. *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 2020, p. 1433–1443.
54. PAPINENI, Kishore; Salim ROUKOS; Todd WARD; Wei-Jing ZHU. Bleu: a method for automatic evaluation of machine translation. *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 2002, p. 311–318.

55. LIN, Chin-Yew. Rouge: A package for automatic evaluation of summaries. *Text summarization branches out*. 2004, p. 74–81.
56. BANERJEE, Satanjeev; Alon LAVIE. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*. 2005, p. 65–72.
57. CASSANO, Federico ir kt. MultiPL-E: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*. 2023.
58. GUO, Daya ir kt. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196*. 2024.
59. LI, Raymond ir kt. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*. 2023.
60. *Big Code Models Leaderboard - a Hugging Face Space by bigcode — huggingface.co* [`https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard`]. [S. a.]. [Accessed 27-05-2024].
61. *OpenAPI Diff — oasdiff.com* [`https://www.oasdiff.com/`]. [S. a.]. [Accessed 24-03-2024].
62. *GitHub - APIs-guru/asyncapi-directory: Directory of asynchronous API specifications in AsyncAPI format — github.com* [`https://github.com/APIs-guru/asyncapi-directory`]. [S. a.]. [Accessed 26-03-2024].
63. *bigcode/the-stack · Datasets at Hugging Face — huggingface.co* [`https://huggingface.co/datasets/bigcode/the-stack`]. [S. a.]. [Accessed 26-03-2024].
64. *GitHub - huggingface/transformers: Transformers: State-of-the-art Machine Learning for Pytorch, TensorFlow, and JAX. — github.com* [`https://github.com/huggingface/transformers`]. [S. a.]. [Accessed 19-05-2024].
65. *GitHub - huggingface/peft: PEFT: State-of-the-art Parameter-Efficient Fine-Tuning. — github.com* [`https://github.com/huggingface/peft`]. [S. a.]. [Accessed 28-05-2024].
66. *Inference Endpoints - Hugging Face — huggingface.co* [`https://huggingface.co/inference-endpoints/dedicated`]. [S. a.]. [Accessed 26-03-2024].
67. *GitHub - belladoreai/llama-tokenizer-js: JS tokenizer for LLaMA 1 and 2 — github.com* [`https://github.com/belladoreai/llama-tokenizer-js`]. [S. a.]. [Accessed 19-05-2024].
68. *BohdanPetryshyn/openapi-completion-refined · Datasets at Hugging Face — huggingface.co* [`https://huggingface.co/datasets/BohdanPetryshyn/openapi-completion-refined`]. [S. a.]. [Accessed 28-05-2024].
69. *hf-code-llama-infiller — npmjs.com* [`https://www.npmjs.com/package/hf-code-llama-infiller`]. [S. a.]. [Accessed 27-05-2024].

Appendices

1. Training and Evaluation Loss of Fine-tuned Models

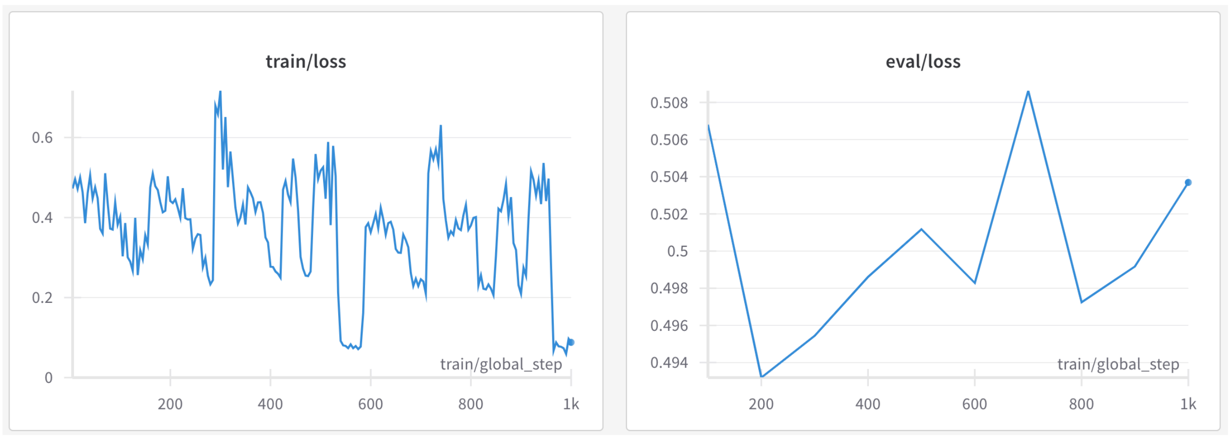


Fig. 18. Training and evaluation loss of the Code Llama 7B model, fine-tuned at 4 096 tokens context size without SPM format

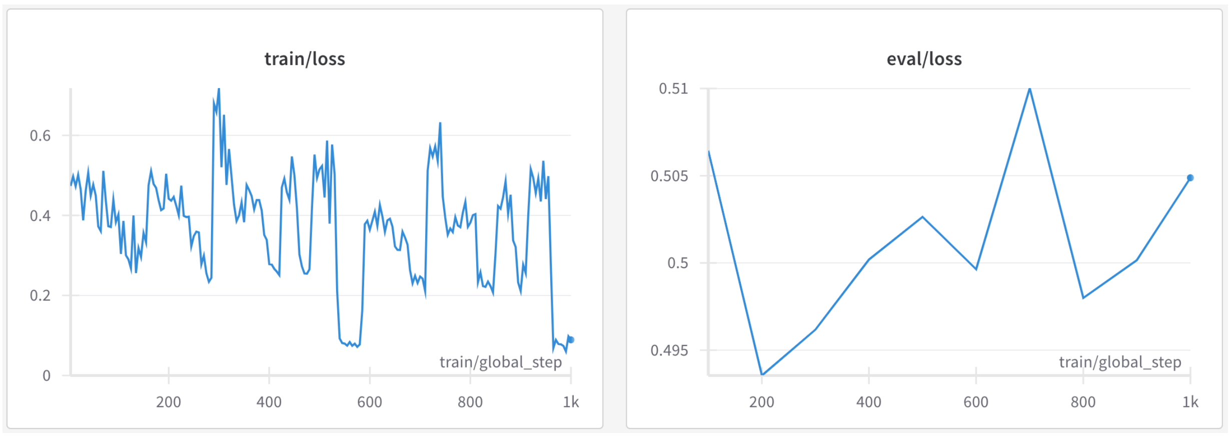


Fig. 19. Training and evaluation loss of the Code Llama 7B model, fine-tuned at 4 096 tokens context size with joint SPM and PSM formats

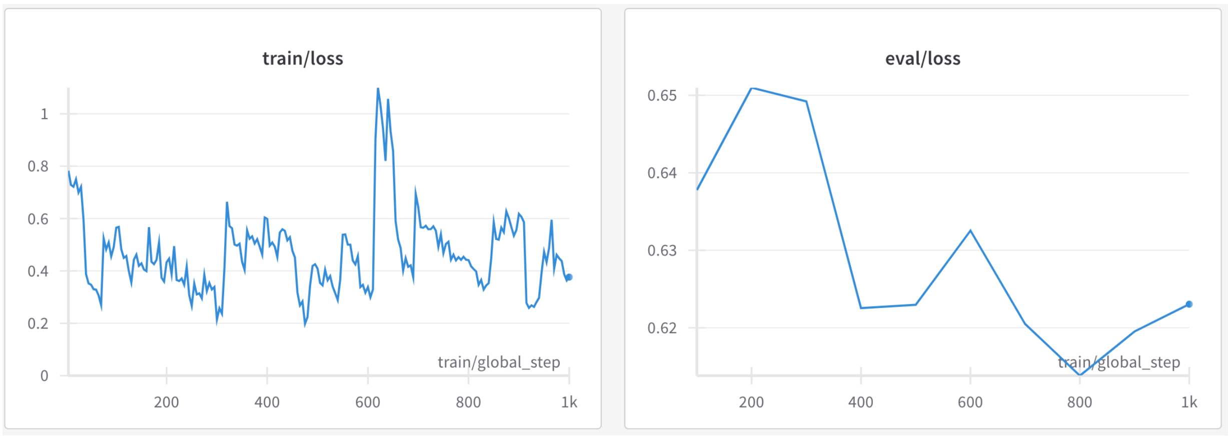


Fig. 20. Training and evaluation loss of the Code Llama 7B model, fine-tuned at 2 048 tokens context size with joint SPM and PSM formats

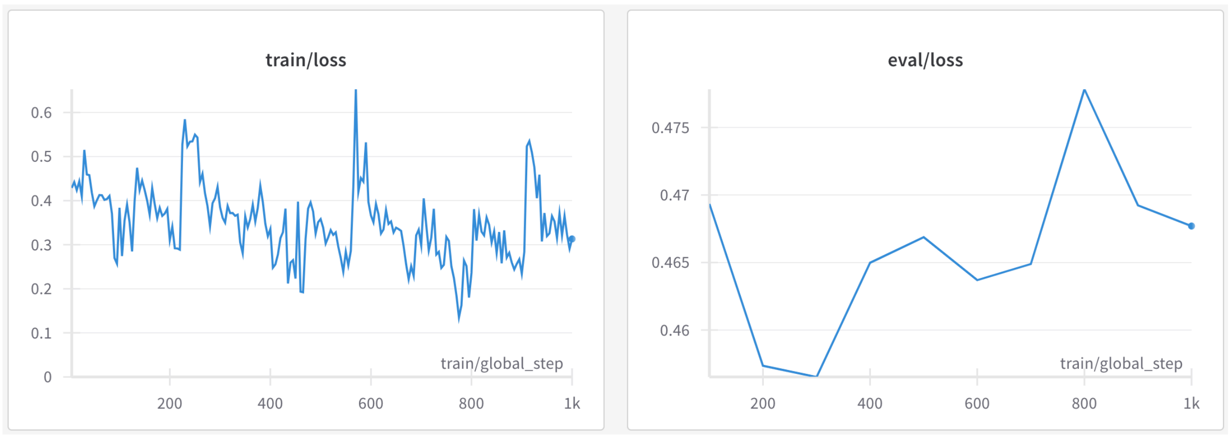


Fig. 21. Training and evaluation loss of the Code Llama 7B model, fine-tuned at 5 120 tokens context size with joint SPM and PSM formats

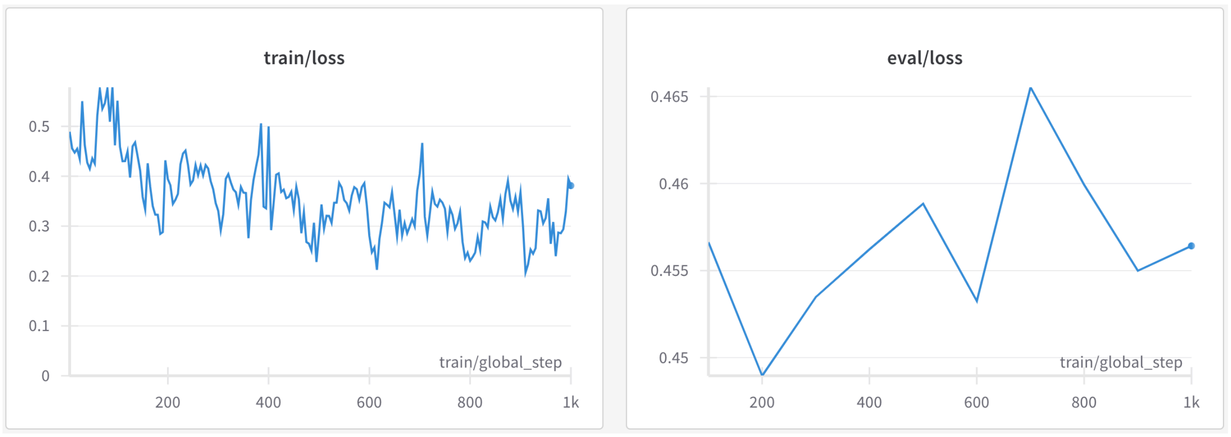


Fig. 22. Training and evaluation loss of the Code Llama 7B model, fine-tuned at 5 120 tokens context size with joint SPM and PSM formats and document splitting technique

2. Example of a Masked OpenAPI Definition used in the Benchmark

```yaml
operationId: app.payments.space.download
parameters:
  - $ref: "#/components/parameters/appId"
  - in: query
    name: ids
    required: true
    schema:
      items:
        type: integer
      type: array
  - description: Optional array with amounts
    in: query
    name: amounts
    schema:
      items:
        type: integer
      type: array
  - in: query
    name: debtorName
    schema:
      type: string
  - in: query
    name: debtorIban
    schema:
      type: string
  - in: query
    name: debtorBic
    schema:
      type: string
responses:
  "200":
    content:
      application/xml:
        schema:
          format: binary
          type: string
    description: Success
  "400":
    description: Bad request
  "404":
    description: Not found
security:
  - {}
```

```
summary: Preview sepa credit transfer file
tags:
  - Payments
```

Listing 1: Example OpenAPI before masking. Highlighted code is selected for masking

```
operationId: app.payments.space.download
parameters:
  - $ref: "#/components/parameters/appId"
  - in: query
    name: ids
    required: true
    schema:
      items:
        type: integer
      type: array
  - description: Optional array with amounts
    in: query
    name: amounts
    schema:
      items:
        type: integer
      type <<MASK>>
    name: debtorBic
    schema:
      type: string
responses:
  "200":
    content:
      application/xml:
        schema:
          format: binary
          type: string
    description: Success
  "400":
    description: Bad request
  "404":
    description: Not found
security:
  - {}
summary: Preview sepa credit transfer file
tags:
  - Payments
```

Listing 2: Example masked OpenAPI definition