Dataflow-Guided Retrieval Augmentation for Repository-Level Code Completion

Anonymous ACL submission

Abstract

Recent years have witnessed the deployment of code language models (LMs) in various code intelligence tasks such as code completion. Yet, it is challenging for pre-trained LMs to generate correct completions in private repositories. Previous studies retrieve cross-file context based on import relations or text similarity, which is insufficiently relevant to completion targets. In this paper, we propose a dataflow-guided retrieval augmentation approach, called DRACO, for repository-level code completion. DRACO parses a private repository into code entities and establishes their relations through an extended dataflow analysis, forming a repo-specific context graph. Whenever triggering code completion, DRACO precisely retrieves relevant background knowledge from the repo-specific context graph and generates well-formed prompts to query code LMs. Furthermore, we construct a large Python dataset, ReccEval, with more diverse completion targets. Our experiments demonstrate the superior accuracy and applicable efficiency of DRACO, improving code exact match by 3.43% and identifier F1-score by 3.27% on average compared to the state-of-the-art approach.

1 Introduction

Pre-trained language models (LMs) of code (Chen et al., 2021; Nijkamp et al., 2023a,b; Allal et al., 2023; Li et al., 2023b) have shown remarkable performance in improving programming productivity (Kazemitabaar et al., 2023; Dakhel et al., 2023). Instead of using a single code file, well-designed programs emphasize separating complicated functionality into independent modules (Barnett and Constantine, 1968). While facilitating collaborative development and software maintenance, it introduces the real-world problem of *repository-level code completion*: given an unfinished code file in a private repository, complete the following pieces of code at the cursor position.

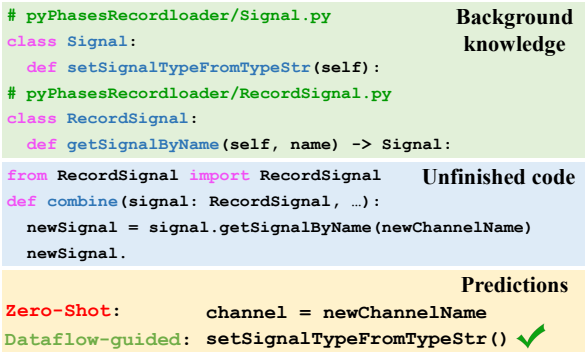

Figure 1: A real-world example of repository-level code completion. The code LM CodeGen25-7B-mono fails to complete the last code line correctly when entering only the unfinished code (Zero-Shot). The model needs background knowledge relevant to `newSignal`, and the retrieval of this knowledge can be guided by dataflow.

Despite pre-training on large-scale corpora, code LMs are still blind to unique naming conventions and programming styles in private repositories (Pei et al., 2023; Liu et al., 2023b; Ding et al., 2023). Previous works finetune LMs to leverage cross-file context (Ding et al., 2022; Shrivastava et al., 2023a,b), which requires additional training data and is difficult to work with larger LMs. Recently, retrieval-augmented generation (RAG) is widely used to aid pre-trained LMs with external knowledge and maintain their parameters intact (Lewis et al., 2020; Mallen et al., 2023; Trivedi et al., 2023). For repository-level code completion, the retrieval database is the current private repository. The state-of-the-art approach, RepoCoder (Zhang et al., 2023), iteratively incorporates a text similarity-based retriever and a code LM.

As shown in Figure 1, the CodeGen25 Python model (Nijkamp et al., 2023a) with 7 billion parameters assigns a value to the attribute `channel` of the object `newSignal`, which seems rational in the unfinished code but is actually outside the list of valid attributes. Due to the lack of similar code snippets in the repository, the text similarity-based approach (Zhang et al., 2023) also fails to com-

plete the correct code line. From a programmer's perspective, one would explore the data origin of the variable `newSignal` in the last line. It comes from the call `signal.getSignalByName`, where the variable type of `signal` is `RecordSignal` imported from the module `RecordSignal`. After providing relevant background knowledge in the private repository, the model would know that the variable type of `newSignal` is the class `Signal` and thus call the correct function.

Inspired by this programming behavior in private repositories, we propose DRACO, a novel dataflow-guided retrieval augmentation approach for repository-level code completion, which steers code LMs with relevant background knowledge rather than similar code snippets. Dataflow analysis is a static program analysis reacting to data dependency relations between variables in a program. In this work, we extend traditional dataflow analysis by setting type-sensitive dependency relations. We employ the standard RAG framework (Lewis et al., 2020): (i) *Indexing*, which parses a private repository into code entities and establishes their relations through dataflow analysis, forming a repo-specific context graph for retrieval. (ii) *Retrieval*, which uses dataflow analysis to obtain fine-grained import information in the unfinished code and retrieves relevant code entities from the pre-built context graph. (iii) *Generation*, which organizes the relevant background knowledge as natural code and concatenates it with the unfinished code to generate well-formed prompts for querying code LMs.

In addition to the existing dataset CrossCodeEval (Ding et al., 2023) for repository-level code completion, we construct a new dataset, ReccEval, with diverse completion targets collected from Python Package Index (PyPI).[1] We conduct experiments with popular LMs including adapted code LMs (Rozière et al., 2023), specialized code LMs (Nijkamp et al., 2023a,b; Allal et al., 2023; Li et al., 2023b), and GPT models (Ouyang et al., 2022; OpenAI, 2023). Our experiments demonstrate that DRACO achieves generally superior accuracy across all settings. Furthermore, DRACO is plug-and-play for various code LMs and efficient to real-time code completion.

Our main contributions are outlined as follows:

- We design an extended dataflow analysis by setting type-sensitive data dependency relations, which supports more precise retrieval.
- We propose DRACO,[2] a dataflow-guided retrieval augmentation approach for repository-level code completion. DRACO builds a repo-specific context graph for retrieval and generates well-formed prompts with relevant background knowledge in real-time completion.
- We construct a Python dataset ReccEval with diverse completion targets. The experimental results show that DRACO improves code exact match by 3.43%, identifier F1-score by 3.27%, and prompt generation time by 100× on average compared to the second-best approach RepoCoder (Zhang et al., 2023).

2 Related Work

Code completion. Early studies adopt statistical LMs (Raychev et al., 2014; Proksch et al., 2015; Raychev et al., 2016; He et al., 2021) and neural models (Li et al., 2018; Svyatkovskiy et al., 2019; Kim et al., 2021; Izadi et al., 2022; Tufano et al., 2023) for code completion. After pre-training on large-scale code corpora, code LMs are familiar with frequent code patterns and achieve superior performance (Chen et al., 2021; Lu et al., 2021; Wang et al., 2021; Le et al., 2022; Allal et al., 2023; Li et al., 2023b; Nijkamp et al., 2023a,b; Shen et al., 2023; Zheng et al., 2023). Unlike single-file code completion, repository-level code completion draws much attention to practical development. Ding et al. (2022) learn in-file and cross-file context jointly on top of pre-trained LMs. Lu et al. (2022) present ReACC to train a code-to-code search retriever and a code completion generator with an external source code database. Shrivastava et al. (2023b) generate example-specific prompts using a prompt proposal classifier and further propose RepoFusion (Shrivastava et al., 2023a) to incorporate relevant repository context by training code LMs. RepoCoder (Zhang et al., 2023) is an iterative retrieval-generation framework to approximate the intended completion target. Despite their good performance, these methods are limited by the high overhead of extra training or iterative generation.

Retrieval-augmented generation. For the scenarios where required knowledge is missing or outdated in pre-trained LMs, RAG has achieved state-of-the-art performance in many NLP tasks (Cai et al., 2022; Feng et al., 2023; Mallen et al., 2023). Usually, RAG integrates the retrieved knowledge

[1] https://pypi.org/

[2] Source code and datasets are submitted on OpenReview.

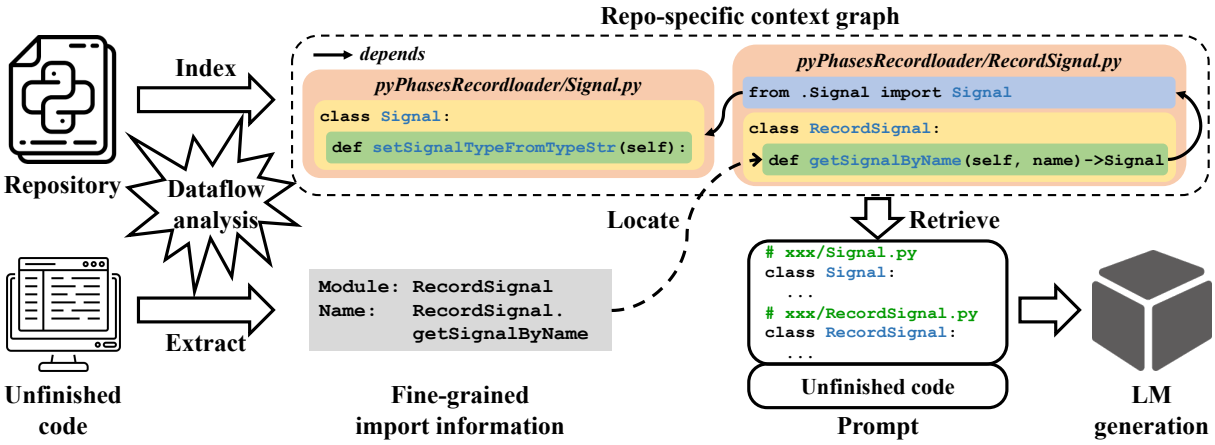

Figure 2: Overview of our approach, where dataflow analysis is crucial for both indexing and retrieval. The details of the unfinished code have been shown in Figure 1. The rectangular boxes visualize *contains* relations between the code entities in the repo-specific context graph, and the solid arrows indicate *depends* relations.

with frozen pre-trained LMs (Ram et al., 2023; Levine et al., 2022; Shi et al., 2023). There exist different types of retrievals including term-based sparse retriever (Robertson and Zaragoza, 2009; Trivedi et al., 2023), embedding-based dense retriever (Karpukhin et al., 2020; Lewis et al., 2020), commercial search engines (Nakano et al., 2021; Liu et al., 2023c), and LMs themself (Yu et al., 2023; Sun et al., 2023). RAG is also broadly applied to code intelligence tasks such as code summarization (Liu et al., 2021; Zhang et al., 2020; Zhou et al., 2023) and code generation (Hashimoto et al., 2018; Parvez et al., 2021; Li et al., 2023a). In this work, we leverage dataflow analysis to guide retrieval, which mines more precise data dependency information for repository-level code completion.

3 Methodology

As shown in Figure 2, DRACO employs the standard RAG framework (Lewis et al., 2020) including indexing (§3.2), retrieval (§3.3), and generation (§3.4). Since our extended dataflow analysis is throughout DRACO, we first introduce it in §3.1. In this work, we focus on Python and the task of single-line code completion, which simulates real-world scenarios where users are programming in integrated development environments (IDEs) and only the context before the cursor is visible.

3.1 Dataflow Analysis

Dataflow analysis is a static program analysis that reacts to the data dependency relations between variables in a program, producing a dataflow graph (DFG). It provides code semantic information that is not affected by personal naming conventions and programming styles.

We assume that the background knowledge relevant to variable types is crucial for code completion. Take the statement `v = f(p)` as an example, the parameter `p` has far less influence on the variable `v` than the call `f` does. Therefore, we extend traditional dataflow analysis by setting dependency relation types. We focus on five *type-sensitive relations*, which indicate what the variable type is or where it derives from:

- *Assigns* relation is a one-to-one correspondence in an assignment statement, which controls variable creation and mutation.
- *As* relation is from *with* or *except* statements and similar with the *assigns* relation.
- *Refers* relation represents a reference to an existing variable or its attribute.
- *Typeof* relation is from the explicit type hints (van Rossum and Lehtosalo, 2022) written by programmers, indicating the data type of the (return) value of a variable or function.
- *Inherits* relation is an implicit data dependency relation since a subclass inherits all the class members of its base classes.

Our DFG is a heterogeneous directed acyclic graph $G = \{(h, r, t) \mid h, t \in E, r \in R\}$, where E denotes the entity set, R denotes the type-sensitive relation set, and a triplet (h, r, t) represents the head entity h pointing to the tail entity t with the relation r. The details of our DFG construction are shown in Appendix A.

3.2 Repo-specific Context Graph

Offline preprocessing is often used in RAG to index a retrieval database. Instead of treating source code as text (Lu et al., 2022; Zhang et al., 2023), we parse a private repository into code entities and es-

tablish their relations through our dataflow analysis, forming a repo-specific context graph.

For each code file in a repository, we traverse its abstract syntax tree (AST) to collect code entities including modules, classes, functions, and variables. A module entity stores its file path and docstring as properties. A class entity stores its name, signature, docstring, and starting line number. A function entity stores its name, signature, docstring, body, and starting line number. A variable entity stores its name, statement, and starting line number. There are natural *contains* relations between these entities, e.g., the class `RecordSignal` *contains* its member function `getSignalByName`. Based on the type-sensitive relations in DFG, we establish *depends* relations between the entity pairs in individual modules, e.g., `Signal` is the return type of the function `getSignalByName`. Eventually, we establish *depends* relations between the variables in local `import` statements and the pointing entities in other modules, e.g., the imported `Signal` points to the class `Signal` in another module.

3.3 Dataflow-Guided Retrieval

Given an unfinished code, we identify fine-grained import information by dataflow analysis and retrieve relevant entities from the repo-specific context graph. We do not intend to perform precise type inference (Peng et al., 2022) for a dynamically typed language like Python, but rather provide relevant background knowledge to code LMs, which provides the definitions of code entities such as class members and function arguments.

All cross-file context is indicated by local `import` statements in Python. However, only using such coarse-grained import information may overlook the knowledge of its specific usages (Ding et al., 2022). We denote import information by $(module, name)$, where $module$ indicates another code file in the repository and $name$ indicates the specific code entity. Particularly, $name$ can be expanded by its *refers* relations in the extracted DFG. For example, we would obtain the fine-grained import information $(module, name.attr)$ if there is a code statement containing `name.attr`.

For each local `import` statement, we collect a set of fine-grained import information. Import information points to code entities in the repo-specific context graph, which is achieved through directory structure and string matching. Let us see Figure 2 for example. Given the fine-grained import information (`RecordSignal.getSignalByName`), we first identify the corresponding module entity *pyPhases-Recordloader/RecordSignal.py* through directory structure. Then, the code entity `getSignalByName` in class `RecordSignal` contained in the module is located by string matching. Finally, the relevant entities are retrieved along *depends* relations using a depth-first search. The retrieved entities provide comprehensive type-related background knowledge for both cross-file imports and usages in the unfinished code.

3.4 Prompt Generation

Before querying LMs, we restore the retrieved entities to the source code and concatenate it with the unfinished code to generate well-formed prompts.

As the maximum input lengths of LMs are finite and fixed, we employ the dynamic context allocation strategy in (Shrivastava et al., 2023b). It pre-allocates half of the total input lengths for the relevant background knowledge and the other half for the unfinished code. If either is shorter than the allocated length, the remaining tokens are allocated to the other. We first consider the entities that have data relations with the line to be completed as the most relevant entities, which form the primary prompt. Then, the entities from other local `import` statements are incrementally added to the prompt to help code LMs further understand the context. This process would terminate before the prompt length exceeds the maximum input length, which prevents the primary prompt from being truncated.

Our mission is to organize the prompts like the original code to maintain the nature of programs (Hindle et al., 2012). We group the relevant entities in modules and merge those with *contains* relations to avoid redundancy, e.g., class members would not be duplicated if the class already exists. The code entities in the same module are sorted by their starting line numbers. The modules in prompts are ranked according to three priorities: (i) Dependent modules come first. An `import` statement may involve several modules, which constitutes a topology structure. If an entity in module m_1 points to an entity in module m_2 along *depends* relations, the content of m_2 should be placed in front of that of m_1, which is consistent with programming conventions. (ii) Once there are multiple options in the topology, the relevant modules are placed ahead (Liu et al., 2023a). We consider the modules that have data relations with the line to be completed as relevant modules and set a lower priority for other

local modules. (iii) Among the `import` statements with the same relevance, we prioritize them by their starting line numbers to get deterministic prompts. Furthermore, a comment "# file path of the module" is put ahead of each module to indicate the relative directory structure. Finally, we place the relevant background knowledge inside a multi-line string, which precedes the unfinished code.

Benefiting from the design of our repo-specific context graph, there are two prompt scopes, named *definition* and *complete*, to control the details of code entities. Compared with only definitions, prompts under the *complete* scope contain specific function bodies and variable statements.

4 Experiment Setup

4.1 Datasets

The widely-used datasets (Raychev et al., 2016; Lu et al., 2021; Peng et al., 2023) for code completion only provide a single unfinished code file as input. Several recent benchmarks (Zhang et al., 2023; Liu et al., 2023b) evaluate next-line prediction, which is different from our concern with the current incomplete line. CrossCodeEval (Ding et al., 2023) is a multilingual benchmark for repository-level code completion, where the statement to be completed has at least one use of cross-file API. Since we focus on Python, we evaluate our DRACO on the Python subset of CrossCodeEval.

We further build a new Python dataset ReccEval with more diverse completion targets. See Appendix B for details. The statistics of ReccEval and the Python subset of CrossCodeEval are shown in Table 1, where the number of tokens is calculated using the StarCoder tokenizer (Li et al., 2023b).

4.2 Implementation Details

We evaluate the retrieval-augmented methods that do not involve training, which excludes several works (Shrivastava et al., 2023a,b; Lu et al., 2022). See Appendix C for more details:

- **Zero-Shot** directly feeds the unfinished code to code LMs, which evaluates their performance without any cross-file information.
- **CCFinder** (Ding et al., 2022) is a cross-file context finder tool retrieving the relevant cross-file context from the pre-built project context graph by `import` statements. We conduct experiments for CCFinder-k ($k = 1, 2$), which indicates that CCFinder retrieves k-hop neighbors of cross-file code entities.

Features	CrossCodeEval	ReccEval
# Repositories	471	2,635
# Examples	2,665	6,461
Avg. # files in repository	30.5	24.6
Avg. # lines in input	73.9	113.1
Avg. # tokens in input	938.9	1,296.2
# Last char of input	dot	any
Avg. # tokens in reference	13.2	8.6

Table 1: Statistics of the Python subset of CrossCodeEval and the ReccEval dataset that we construct.

Models		Parameter sizes
Specialized models	CodeGen	350M, 2.7B, 6.1B, 16.1B
	CodeGen2.5	7B
	SantaCoder	1.1B
	StarCoder	15.5B
Adapted model	Code Llama	7B
GPT models	GPT-3.5, GPT-4	-

Table 2: The LMs used in our experiments.

- **RG-1 and RepoCoder** (Zhang et al., 2023) construct a retrieval database through a sliding window and retrieve similar code snippets using text similarity-based retrievers. RepoCoder is an iterative retrieval-generation framework, which retrieves the database with the results generated in the previous iteration. RG-1 represents the standard RAG and is the first iteration of RepoCoder.

As shown in Table 2, we conduct comprehensive experiments on seven popular LMs. See Appendix D for details. For a method, we first preprocess all repositories in the datasets. Then, we generate prompts for the unfinished code and record the time used. Finally, we acquire the completion results by feeding the prompts to each code LM. A prediction is the first line of a completion result.

4.3 Evaluation Metrics

We evaluate the accuracy of each method by code match and identifier match scores (Ding et al., 2023), as well as the efficiency by prompt generation time. We report the average of each metric:

- **Code match.** Given a prediction y and the reference y^*, we assess y using the exact match accuracy (EM) and the Levenshtein edit similarity (ES) (Lu et al., 2021; Zhang et al., 2023). EM is calculated by an indicator function whose value is 1 if $y = y^*$; otherwise, it is 0. ES $= 1 - \frac{\text{Lev}(y,y^*)}{\max(||y||,||y^*||)}$, where $||\cdot||$ calculates the string length and Lev() calculates the Levenshtein distance.
- **Identifier match.** Identifier exact match

Methods	CodeGen-350M				SantaCoder-1.1B				CodeGen25-7B				StarCoder-15.5B			
	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1
Zero-Shot	2.81	55.01	8.22	38.02	3.79	57.92	10.43	41.98	7.77	60.52	14.45	45.40	8.71	62.08	16.02	47.58
CCFinder-1	9.64	59.05	16.36	45.33	14.37	63.86	22.89	52.26	18.84	66.67	27.35	56.05	27.99	72.59	38.24	64.46
CCFinder-2	8.22	58.17	14.52	44.15	11.41	62.47	19.74	49.90	15.50	65.27	24.05	53.56	28.67	73.25	39.10	65.59
RG-1	9.19	60.10	16.89	46.45	12.35	64.09	22.10	51.79	17.34	67.36	27.28	56.22	26.27	72.70	37.00	64.04
RepoCoder	10.13	61.25	18.65	48.29	13.62	65.53	23.94	54.06	19.51	68.98	29.57	58.51	29.12	74.56	40.83	66.81
DRACO	**13.02**	**61.30**	**20.53**	**49.04**	**20.64**	**67.04**	**29.83**	**57.37**	**24.99**	**70.10**	**34.63**	**61.14**	**34.67**	**75.83**	**45.63**	**69.93**

Table 3: Performance comparison on the CrossCodeEval dataset. Numbers are shown in percentage (%).

Methods	CodeGen-350M				SantaCoder-1.1B				CodeGen25-7B				StarCoder-15.5B			
	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1
Zero-Shot	4.01	49.41	9.75	25.98	5.54	52.95	11.93	29.94	11.10	57.25	17.37	35.55	12.77	58.84	20.03	38.12
CCFinder-1	14.15	55.75	21.24	37.74	21.36	61.90	29.31	46.18	26.87	65.76	34.55	51.00	39.33	73.05	48.18	63.49
CCFinder-2	11.64	53.70	17.94	34.15	17.12	59.57	24.58	41.93	22.49	63.42	29.72	46.81	39.92	73.29	48.91	64.08
RG-1	19.44	59.08	26.02	40.92	23.62	63.23	30.58	46.24	29.33	66.94	36.06	51.36	42.67	74.64	51.11	64.64
RepoCoder	**22.46**	**60.59**	29.05	43.91	27.29	65.06	34.56	49.68	32.84	68.73	40.07	54.73	46.26	76.44	54.47	67.59
DRACO	22.12	60.41	**29.73**	**46.09**	**30.26**	**66.90**	**39.08**	**55.43**	**36.46**	**70.76**	**44.67**	**60.40**	**46.49**	**76.80**	**55.98**	**70.32**

Table 4: Performance comparison on the ReccEval dataset.

(ID.EM) and F1-score test the model's ability to predict the correct APIs (Ding et al., 2023). We parse the code to extract the identifiers from y and y^* and get two ordered identifier lists, which are used for the two metrics.

- **Prompt generation time.** To evaluate the efficiency of code completion, we record the prompt generation time, which contains the time to retrieve relevant context and the time to assemble final prompts. Note that we ignore the time spent by code LMs in generating predictions, which is determined by the used LMs rather than the methods.

5 Experimental Results and Analysis

5.1 Performance Comparison

The performance comparison on the CrossCodeEval and ReccEval datasets is listed in Tables 3 and 4, respectively. Additional results on other CodeGen models are supplemented in Appendix E.1. Overall, DRACO significantly improves the accuracy of various code LMs. Particularly, the CodeGen-350M model integrated with DRACO even outperforms the zero-shot StarCoder-15.5B model.

In comparison to other retrieval-augmented methods, DRACO also shows generally superior accuracy across all settings. The average absolute improvement on EM, ES, ID.EM, and F1 versus RepoCoder is 3.43%, 1.00%, 3.62%, and 3.27%, respectively. RepoCoder retrieves similar code demonstrations that help increase the ES metric of completion results. But RepoCoder ignores the validity of its generated identifiers in private repositories, which decreases the metrics for code exact match and identifier match. Such almost correct completion results may introduce unconscious bugs to the programmers who are unfamiliar with the repository. In contrast, DRACO presents the definitions of relevant code entities, providing better control over code LMs to generate valid identifiers. Moreover, the background knowledge can be used as a reference to help the programmers understand and review the completion results in IDEs. DRACO using the CodeGen-350M model is slightly worse than RepoCoder in terms of code match metrics on the ReccEval dataset, as the model may not be powerful enough to capture the data relations in our provided background knowledge.

CCFinder retrieves cross-file code entities by plain import relations. The entities retrieved by CCFinder were originally designed to be encoded for training code LMs. When used as a retrieval-augmented method, CCFinder retrieves too many code entities based on coarse-grained import information, resulting in truncation of truly relevant context. As a result, CCFinder-2 with more retrieved entities only exceeds CCFinder-1 slightly on the StarCoder model that supports longer inputs. Therefore, the subsequent analysis experiments are conducted with CCFinder-1. Guiding by our dataflow analysis, DRACO retrieves relevant code entities more precisely, leading to significantly superior accuracy.

The performance of code completion varies on the two datasets. First, according to the statistics in Table 1, the average reference length of ReccEval is significantly shorter than that of CrossCodeEval, leading to the higher EM metrics of both code and identifier on ReccEval. Moreover, all inputs of CrossCodeEval end with a dot where a correct

Methods	CrossCodeEval	ReccEval
CCFinder-1	32	49
CCFinder-2	52	82
RG-1	**13**	**15**
RepoCoder	4,062	4,413
DRACO	40	44

Table 5: Prompt generation time (in milliseconds) of each method using the CodeGen-350M model.

API is required in the first place, which is more suitable for CCFinder and DRACO that retrieve code definitions. Many inputs of ReccEval end with partial names of the target APIs, which facilitates text similarity-based retrievals including RG-1 and RepoCoder. Therefore, the lead of DRACO on CrossCodeEval is more significant.

5.2 Efficiency Evaluation

The time spent on prompt generation is perceived by users whenever code completion is triggered. Table 5 shows the prompt generation time of each method using the CodeGen-350M model, and additional results are shown in Appendix E.2. CCFinder and DRACO require parsing the unfinished code into an AST or a DFG, which is slightly slower than RG-1 with text similarity-based retrieval but still comparable. RepoCoder relies on RG-1 to generate sufficient content for the second retrieval, which results in more than 4 seconds even on the smallest CodeGen-350M model and may not be feasible for real-time code completion.

In summary, DRACO is efficient for real-time code completion in IDEs. Compared to the methods with comparable efficiency (i.e., excluding RepoCoder), DRACO is considerably ahead in the accuracy of repository-level code completion.

5.3 Ablation Study

To analyze the effectiveness of dataflow analysis in DRACO, we conduct an ablation study shown in Tables 6 and 16. "w/o cross_df" disables *depends* relations in the repo-specific context graph, making DRACO unable to handle the data dependency relations in other code files. "w/o intra_df" disables the dataflow analysis for the unfinished code, which only allows DRACO to retrieve coarse-grained import information in the order of their starting line numbers. "w/o dataflow" degenerates DRACO into a naive method that simply takes the imported cross-file entities in the unfinished code as the relevant background knowledge.

The ablation study demonstrates that the complete DRACO achieves the best performance, and

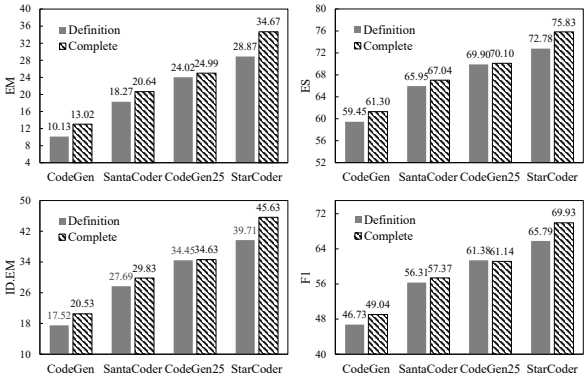

Figure 3: Performance comparison of two prompt scopes on the CrossCodeEval dataset.

all usages of dataflow analysis play a positive role in repository-level code completion. It can be observed that the enhancement of the "intra_df" component on the StarCoder model is less than that on other models. This component places the more relevant background knowledge in front of the prompt to prevent truncation, which is weakened to some extent on the StarCoder model with a maximum context length of 8K tokens.

The accuracy of DRACO without dataflow analysis is still comparable with CCFinder. CCFinder groups the relevant context in code entities, which is counter-intuitive for source code (see the example shown in Appendix F.1). The results reveal that the well-formed prompts generated by DRACO can better steer code LMs, even if the depth-first search for code entities is absent.

5.4 Analysis of Prompt Scopes

The prompts generated by DRACO consist of the definitions of code entities, which provide options for the *definition* and *complete* scopes, as described in Section 3.4. We further conduct experiments to evaluate the influence of the two prompt scopes. The results on the CrossCodeEval and ReccEval datasets are shown in Figures 3 and 7, respectively.

DRACO with the *complete* scope achieves the best performance across all settings, which indicates that code implementation can further enhance LMs. Implementation details can provide a deeper understanding of code entities, along with the programming styles. Moreover, DRACO with the *definition* scope outperforms CCFinder and RG-1 in most settings (cf. Tables 3 and 4), suggesting that the definitions without specific implementations are also useful for code LMs. Since an implementation is usually much longer than its definitions, both prompt scopes are optional in practical applications, in a trade-off between accuracy and cost.

Methods	CodeGen-350M				SantaCoder-1.1B				CodeGen25-7B				StarCoder-15.5B			
	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1
DRACO	**13.02**	**61.30**	**20.53**	**49.04**	**20.64**	**67.04**	**29.83**	**57.37**	**24.99**	**70.10**	**34.63**	**61.14**	**34.67**	**75.83**	**45.63**	**69.93**
w/o cross_df	12.12	60.93	19.51	48.32	18.42	66.05	27.62	55.64	22.59	69.15	31.89	59.36	30.73	73.85	41.05	66.31
w/o intra_df	10.88	59.74	17.56	46.25	15.95	64.11	24.09	52.72	19.59	67.08	28.33	56.14	32.35	74.60	43.00	67.98
w/o dataflow	10.13	59.55	17.00	45.88	14.90	63.57	23.11	51.88	18.57	66.85	27.13	55.53	28.82	72.80	38.87	64.65

Table 6: Ablation study for dataflow analysis on the CrossCodeEval dataset.

Methods	GPT-3.5					GPT-4					StarCoder-15.5B			
	EM	ES	ID.EM	F1	WOF	EM	ES	ID.EM	F1	WOF	EM	ES	ID.EM	F1
Zero-Shot	10.00	57.20	10.00	44.39	0	16.00	67.20	20.00	56.82	0	12.00	65.44	18.00	53.18
CCFinder-1	20.00	66.32	**30.00**	57.12	0	38.00	74.80	46.00	69.36	4	34.00	76.24	46.00	72.31
RG-1	14.00	54.06	20.00	43.06	9	12.00	35.20	18.00	22.30	34	20.00	74.36	38.00	70.08
RepoCoder	18.00	63.44	22.00	57.13	1	34.00	73.18	40.00	65.07	7	26.00	72.48	42.00	68.99
DRACO	**24.00**	**67.54**	**30.00**	**58.08**	0	**42.00**	**76.58**	**50.00**	**72.36**	5	**38.00**	**77.84**	**52.00**	**77.73**

Table 7: Performance comparison on the sampled CrossCodeEval dataset. "WOF" is a manual count indicating the number of predictions with wrong output format, such as "The last line of the code should be:".

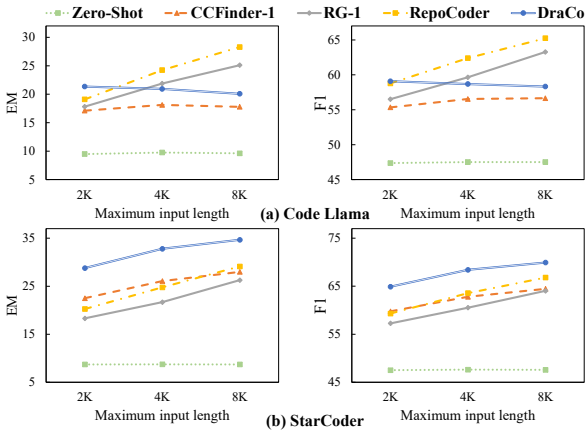

Figure 4: Performance changes with different maximum input lengths on the CrossCodeEval dataset.

5.5 Analysis of Adapted LM

We analyze the effect of different maximum input lengths for different types of LMs. Specifically, we evaluate Code Llama-7B and StarCoder-15.5B with 2K, 4K, and 8K tokens. The performance changes are shown in Figure 4, and the complete results are presented in Tables 17–20.

With the increase of the maximum input length, the accuracy of DRACO applied to Code Llama decreases, which shows the opposite trend of the StarCoder model. Code Llama is created by further training Llama 2 on its code-specific datasets. It is different from specialized code LMs such as StarCoder which are mainly pre-trained on code corpora. With similar background knowledge, Code Llama-7B does not have enough capability to capture data dependency relations in long Python code, leading to the degraded accuracy of DRACO. In contrast, specialized code LMs can understand longer code context and may be a better choice for repository-level code completion.

5.6 Analysis of GPT Models

We randomly sample 50 examples from CrossCodeEval and evaluate them with GPT-3.5, GPT-4, and StarCoder-15.5B. The results shown in Table 7 reveal that: (i) DRACO can also significantly enhance GPT models and achieve superior accuracy. (ii) Code completion with GPT models suffers from the difficulties in output format and API cost. Given the long context of repository-level code completion, there lacks sufficient length to place the demonstrations required for in-context learning (Brown et al., 2020). It is hard to control the output format of GPT models through instruction, which may introduce bias into the evaluation, especially for RG-1 with GPT-4. Moreover, the API cost for this evaluation (only 50 examples) is nearly 35 US dollars. (iii) Excluding the effect of wrong output format, we may assume "GPT-4 > StarCoder-15.5B > GPT-3.5" for this task.

6 Conclusions

This paper proposes DRACO, a dataflow-guided retrieval augmentation approach for repository-level code completion. To guide more precise retrieval, an extended dataflow analysis is designed by setting type-sensitive data dependency relations. DRACO indexes private repositories to form repo-specific context graphs and retrieves relevant background knowledge from them, which is assembled with the unfinished code to generate well-formed prompts for querying code LMs. The experiments on the CrossCodeEval dataset and our ReccEval dataset demonstrate the superior accuracy and applicable efficiency of DRACO. In future work, we will explore other structured code representations.

Ethical Considerations

The code generated by pre-trained LMs may contain non-existent APIs or even introduce potential bugs. The retrieval-augmented approaches including ours mitigate this issue only to some extent. We recommend presenting our retrieved background knowledge to programmers for review and taking appropriate care of these risks if deploying our approach in real-world applications.

All the datasets and code LMs used in this work are publicly available with permissive licenses. The CrossCodeEval dataset and CodeGen family are licensed under the Apache-2.0 License. The SantaCoder and StarCoder models are licensed under the BigCode OpenRAIL-M v1 license agreement. Code Llama is governed by the Meta license.[3] The repositories in our ReccEval dataset are all licensed under permissive licenses including MIT, Apache, and BSD licenses.

Limitations

DRACO relies on a code LM to support long inputs and capture data dependency relations in the provided background knowledge. Thus, the performance of DRACO may be limited by the capability of the code LM. According to our experiments, DRACO still has a considerable improvement on the smallest CodeGen-350M model with 2K tokens, which mitigates this limitation.

The effectiveness of DRACO may degrade when the code intent is unclear. For new line or function body completion, the guidance of dataflow analysis is weakened since DRACO cannot set priorities for import information. We focus on code completion for an incomplete line, which is a realistic and widely used feature in IDEs. Future work may explore the role of dataflow analysis in different completion scenarios.

DRACO requires changes to migrate to other programming languages. Our idea of guiding retrieval with dataflow analysis is not limited to Python. However, due to the different characteristics of programming languages, DRACO needs to extend dataflow analysis for target languages. The variety of static analysis tools for common programming languages provides convenience for implementing multilingual DRACO.

[3] https://github.com/facebookresearch/llama/blob/main/LICENSE

References

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Muñoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy-Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. 2023. SantaCoder: don't reach for the stars! *CoRR*, 2301.03988:1–35.

Tom O. Barnett and Larry L. Constantine. 1968. *Modular Programming: Proceedings of a National Symposium*. Information & Systems Institute, Leipzig, Germany.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *NeurIPS*, pages 1877–1901, Virtual.

Deng Cai, Yan Wang, Lemao Liu, and Shuming Shi. 2022. Recent advances in retrieval-augmented text generation. In *SIGIR*, pages 3417–3419, Madrid, Spain. ACM.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *CoRR*, 2107.03374:1–35.

Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and

Zhen Ming (Jack) Jiang. 2023. GitHub Copilot AI pair programmer: Asset or liability? *J. Syst. Softw.*, 203:111734.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023. CrossCodeEval: A diverse and multilingual benchmark for cross-file code completion. In *NeurIPS*, pages 1–23, New Orleans, LA, USA.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2022. CoCoMIC: Code completion by jointly modeling in-file and cross-file context. *CoRR*, 2212.10007:1–16.

Zhangyin Feng, Weitao Ma, Weijiang Yu, Lei Huang, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2023. Trends in integration of knowledge and large language models: A survey and taxonomy of methods, benchmarks, and applications. *CoRR*, 2311.05876:1–22.

Tatsunori B. Hashimoto, Kelvin Guu, Yonatan Oren, and Percy Liang. 2018. A retrieve-and-edit framework for predicting structured outputs. In *NeurIPS*, pages 10073–10083, Montréal, Canada.

Xincheng He, Lei Xu, Xiangyu Zhang, Rui Hao, Yang Feng, and Baowen Xu. 2021. PyART: Python API recommendation in real-time. In *ICSE*, pages 1634–1645, Madrid, Spain. IEEE.

Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the naturalness of software. In *ICSE*, pages 837–847, Zurich, Switzerland. IEEE.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *CoRR*, 1909.09436:1–6.

Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. 2022. CodeFill: Multi-token code completion by jointly learning from structure and naming sequences. In *ICSE*, pages 401–412, Pittsburgh, PA, USA. ACM.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781, Online. Association for Computational Linguistics.

Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J. Ericson, David Weintrop, and Tovi Grossman. 2023. Studying the effect of AI code generators on supporting novice learners in introductory programming. In *CHI*, pages 455:1–455:23, Hamburg, Germany. ACM.

Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers. In *ICSE*, pages 150–162, Madrid, Spain. IEEE.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The Stack: 3 TB of permissively licensed source code. *CoRR*, 2211.15533:1–27.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu-Hong Hoi. 2022. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *NeurIPS*, pages 1–15, New Orleans, LA, USA. Curran Associates Inc.

Yoav Levine, Itay Dalmedigos, Ori Ram, Yoel Zeldes, Daniel Jannai, Dor Muhlgay, Yoni Osin, Opher Lieber, Barak Lenz, Shai Shalev-Shwartz, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. 2022. Standing on the shoulders of giant frozen language models. *CoRR*, 2204.10019:1–19.

Patrick S. H. Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *NeurIPS*, pages 9459–9474, Virtual.

Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. 2023a. AceCoder: Utilizing existing code to enhance code generation. *CoRR*, 2303.17780:1–12.

Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code completion with neural attention and pointer networks. In *IJCAI*, pages 4159–4165, Stockholm, Sweden. ijcai.org.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm

de Vries. 2023b. StarCoder: may the source be with you! *CoRR*, 2305.06161:1–54.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023a. Lost in the middle: How language models use long contexts. *CoRR*, 2307.03172:1–18.

Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2021. Retrieval-augmented generation for code summarization via hybrid GNN. In *ICLR*, Virtual. OpenReview.net.

Tianyang Liu, Canwen Xu, and Julian J. McAuley. 2023b. RepoBench: Benchmarking repository-level code auto-completion systems. *CoRR*, 2306.03091:1–18.

Xiao Liu, Hanyu Lai, Hao Yu, Yifan Xu, Aohan Zeng, Zhengxiao Du, Peng Zhang, Yuxiao Dong, and Jie Tang. 2023c. WebGLM: Towards an efficient web-enhanced question answering system with human preferences. In *KDD*, pages 4549–4560, Long Beach, CA, USA. ACM.

Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A retrieval-augmented code completion framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6227–6240, Dublin, Ireland. Association for Computational Linguistics.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *NeurIPS*, pages 1–16, Virtual. Curran Associates, Inc.

Alex Mallen, Akari Asai, Victor Zhong, Rajarshi Das, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. When not to trust language models: Investigating effectiveness of parametric and non-parametric memories. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9802–9822, Toronto, Canada. Association for Computational Linguistics.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. 2021. WebGPT: Browser-assisted question-answering with human feedback. *CoRR*, 2112.09332:1–32.

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023a. CodeGen2: Lessons for training LLMs on programming and natural languages. *CoRR*, 2305.02309:1–12.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023b. CodeGen: An open large language model for code with multi-turn program synthesis. In *ICLR*, pages 1–25, Kigali, Rwanda. OpenReview.net.

OpenAI. 2023. GPT-4 technical report. https://cdn.openai.com/papers/gpt-4.pdf.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *NeurIPS*, volume 35, pages 27730–27744. Curran Associates, Inc.

Md Rizwan Parvez, Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2719–2734, Punta Cana, Dominican Republic. Association for Computational Linguistics.

Hengzhi Pei, Jinman Zhao, Leonard Lausen, Sheng Zha, and George Karypis. 2023. Better context makes better code language models: A case study on function call argument completion. In *AAAI*, pages 5230–5238, Washington, DC, USA. AAAI Press.

Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael R. Lyu. 2022. Static inference meets deep learning: A hybrid type inference approach for Python. In *ICSE*, pages 2019–2030, Pittsburgh, PA, USA. ACM.

Yun Peng, Shuqing Li, Wenwei Gu, Yichen Li, Wenxuan Wang, Cuiyun Gao, and Michael R. Lyu. 2023. Revisiting, benchmarking and exploring API recommendation: How far are we? *IEEE Trans. Softw.*, 49(4):1876–1897.

Sebastian Proksch, Johannes Lerch, and Mira Mezini. 2015. Intelligent code completion with Bayesian networks. *ACM Trans. Softw. Eng. Methodol.*, 25(1):3:1–3:31.

Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. 2023. In-context retrieval-augmented language models. *CoRR*, 2302.00083:1–15.

Veselin Raychev, Pavol Bielik, and Martin T. Vechev. 2016. Probabilistic model for code with decision trees. In *OOPSLA*, pages 731–747, Amsterdam, The Netherlands. ACM.

Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. *ACM SIGPLAN Notices*, 49(6):419–428.

Stephen Robertson and Hugo Zaragoza. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open foundation models for code. *CoRR*, 2308.12950:1–48.

Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, Yuenan Guo, and Qianxiang Wang. 2023. PanGu-Coder2: Boosting large language models for code with ranking feedback. *CoRR*, 2307.14936:1–15.

Weijia Shi, Sewon Min, Michihiro Yasunaga, Minjoon Seo, Rich James, Mike Lewis, Luke Zettlemoyer, and Wen-tau Yih. 2023. REPLUG: Retrieval-augmented black-box language models. *CoRR*, 2301.12652:1–12.

Disha Shrivastava, Denis Kocetkov, Harm de Vries, Dzmitry Bahdanau, and Torsten Scholak. 2023a. RepoFusion: Training code models to understand your repository. *CoRR*, 2306.10998:1–15.

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023b. Repository-level prompt generation for large language models of code. In *ICML*, pages 31693–31715, Honolulu, HI, USA. PMLR.

Zhiqing Sun, Xuezhi Wang, Yi Tay, Yiming Yang, and Denny Zhou. 2023. Recitation-augmented language models. In *ICLR*, Kigali, Rwanda. OpenReview.net.

Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. 2019. Pythia: AI-assisted code completion system. In *KDD*, pages 2727–2735, Anchorage, AK, USA. ACM.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, 2307.09288:1–77.

Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. 2023. Interleaving retrieval with chain-of-thought reasoning for knowledge-intensive multi-step questions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 10014–10037, Toronto, Canada. Association for Computational Linguistics.

Rosalia Tufano, Luca Pascarella, and Gabriele Bavota. 2023. Automating code-related tasks through transformers: The impact of pre-training. In *ICSE*, pages 2425–2437, Melbourne, Australia. IEEE.

Łukasz Langa Guido van Rossum and Jukka Lehtosalo. 2022. PEP 484 – type hints. https://peps.python.org/pep-0484/.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Wenhao Yu, Dan Iter, Shuohang Wang, Yichong Xu, Mingxuan Ju, Soumya Sanyal, Chenguang Zhu, Michael Zeng, and Meng Jiang. 2023. Generate rather than retrieve: Large language models are strong context generators. In *ICLR*, Kigali, Rwanda. OpenReview.net.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *EMNLP*, pages 2471–2484, Singapore. Association for Computational Linguistics.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *ICSE*, pages 1385–1397, Seoul, South Korea. ACM.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A pre-trained model for code generation with multilingual benchmarking on HumanEval-X. In *KDD*, pages 5673–5684, Long Beach, CA, USA. ACM.

Ziyi Zhou, Huiqun Yu, Guisheng Fan, Zijie Huang, and Kang Yang. 2023. Towards retrieval-based neural code summarization: A meta-learning approach. *IEEE Trans. Software Eng.*, 49(4):3008–3031.

A Dataflow Graph Construction

We first parse the Python code into an AST by tree-sitter,[4] which is feasible to parse incomplete code snippets. Based on the AST, we extract variables as the entity set and identify type-sensitive relations as triplets, forming our DFG. The implementation involves specific Python syntax features and is presented in our published code.

The type-sensitive relations are listed in Table 8. We also introduce an example to visualize our DFG construction, as shown in Figure 5. The extracted variables include identifiers (e.g., `newSignal`) and attributes (e.g., `signal.getSignalByName`). The type-sensitive relations are identified as described in Section 3.1. For example, the parameter `signal: RecordSignal` indicates a triplet (`RecordSignal`, *Typeof*, `signal`). The assignment statement forms a triplet (`signal.getSignalByName`, *Assigns*, `newSignal`), and the parameter `newChannelName` is a type-insensitive relation pruned in our DFG.

B Details of Dataset Construction

We collect the projects that are first released on PyPI between 2023-01-01 to 2023-04-28, which is after the releases of pre-training corpora (Husain et al., 2019; Chen et al., 2021; Kocetkov et al., 2022). We pick the projects with permissive licenses (i.e., MIT, Apache, and BSD) and filter out those that have fewer than 6 or more than 100 Python code files. We identify the usages of local imported resources and randomly select a subsequent token as the cursor position. The context before the cursor is the input, while the current line after the cursor is the reference. For the diversity of ReccEval, we limit the maximum number of examples to one per code file and 10 per repository. Moreover, we ensure that the reference is not in the unfinished code and feed the examples to StarCoderBase-1B model (Li et al., 2023b) to remove the exact matches (Ding et al., 2023), which excludes strong clues in the unfinished code to make ReccEval more challenging.

C Implementation Details of Baselines

We describe more implementation details of CCFinder, RG-1, and RepoCoder, which are in line with the experiment setup in their papers:

- **CCFinder.** Because CCFinder is not open source, we reproduce it according to its paper. We do not limit the number of retrieved code entities, as the cross-file context would be truncated if it exceeds the maximum length. We also re-order the retrieved entities, ensuring the entities from the same source file follow the original code order.
- **RG-1 and RepoCoder.** In our experiments, we use a sparse bag-of-words model as their retriever, which calculates text similarity using the Jaccard index and achieves equivalent performance to the dense retriever. The line length of the sliding window and the sliding size are set to 20 and 10, respectively. According to the maximum input length of code LMs, the maximum number of the retrieved code snippets in prompts is set to 40 for the StarCoder model and 10 for other models. The number of iterations of RepoCoder is set to 2.

[4] https://github.com/tree-sitter/tree-sitter

Relations	Examples	Triplets
assigns	`v = u`	(u, *assigns*, v)
as	`with f() as v`	(f, *as*, v)
refers	`u.v`	(u, *refers*, u.v)
typeof	`def f() -> v`	(v, *typeof*, f)
inherits	`class v(u)`	(u, *inherits*, v)

Table 8: Illustrations of type-sensitive relations.

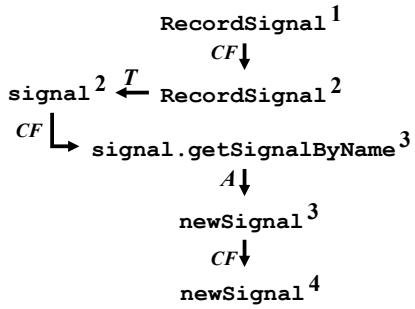

Figure 5: An example of our DFG, which corresponds to the unfinished code in Figure 1. The numbers labeled in the DFG correspond to the line numbers of the variables. The labels on the edges are the initials of the relation names defined in Section 3.1.

D Details of Used LMs

We categorize the used LMs into specialized code LMs, adapted code LMs, and GPT models. The details are listed as follows:

- **CodeGen** (Nijkamp et al., 2023a,b) is a family of auto-regressive LMs for program synthesis. We use the CodeGen2.5 model with 7B parameters and the CodeGen models with 350M, 2.7B, 6.1B, and 16.1B parameters, which support a maximum context length of 2,048 (2K) tokens. We use their mono versions, which are further trained on additional Python tokens.

Methods	CodeGen-2.7B				CodeGen-6.1B				CodeGen-16.1B			
	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1
Zero-Shot	5.44	57.85	11.71	42.22	6.57	59.01	13.13	44.11	7.05	59.88	13.88	45.27
CCFinder-1	14.30	63.18	22.51	51.28	16.21	65.00	24.58	53.70	17.19	65.57	26.19	55.36
CCFinder-2	11.41	61.74	19.47	48.92	13.21	63.23	21.39	51.17	14.15	63.89	22.59	52.17
RG-1	12.68	63.87	21.58	51.89	14.82	65.12	23.53	53.54	15.27	65.87	24.65	54.76
RepoCoder	14.07	65.12	23.90	53.33	15.87	66.74	26.15	55.80	17.04	67.69	27.62	57.36
DRACO	**18.99**	**65.52**	**27.50**	**55.07**	**22.36**	**68.06**	**31.37**	**58.60**	**22.78**	**68.09**	**32.08**	**59.40**

Table 9: Performance comparison on the CrossCodeEval dataset using other CodeGen models (cf. Table 3).

Methods	CodeGen-2.7B				CodeGen-6.1B				CodeGen-16.1B			
	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1
Zero-Shot	6.73	53.30	13.05	30.65	8.34	54.77	14.64	32.60	10.12	55.84	16.50	34.17
CCFinder-1	20.38	60.80	28.12	44.83	23.56	63.07	31.56	47.90	24.64	64.17	32.66	49.28
CCFinder-2	17.21	59.13	24.32	41.58	19.66	60.77	26.93	43.73	20.83	61.85	28.25	45.11
RG-1	24.49	63.12	31.34	46.51	25.86	64.75	32.66	48.37	27.97	66.18	35.07	50.37
RepoCoder	27.84	65.07	35.13	49.71	29.45	66.62	36.71	51.67	31.73	67.94	38.96	53.64
DRACO	**29.42**	**65.91**	**37.63**	**53.69**	**32.05**	**67.93**	**40.83**	**56.80**	**33.76**	**69.20**	**42.38**	**58.38**

Table 10: Performance comparison on the ReccEval dataset using other CodeGen models (cf. Table 4).

- **SantaCoder** (Allal et al., 2023) is a 1.1B model trained on Python, Java, and JavaScript, which supports a maximum context length of 2K tokens.
- **StarCoder** (Li et al., 2023b) is a 15.5B model trained on 80+ programming languages and further trained on Python, which supports a maximum context length of 8K tokens.
- **Code Llama** (Rozière et al., 2023) is created by further training Llama 2 (Touvron et al., 2023) on its code-specific datasets, which supports a maximum context length of 16k tokens. Considering GPU member and efficiency, we chose the 7B Python specialist version named CodeLlama-7b-Python-hf and limit the maximum context length to 8K tokens.
- **GPT models** (Ouyang et al., 2022; OpenAI, 2023) are commercial black box models released by OpenAI. The used API of GPT-3.5 is gpt-3.5-turbo-0613 with a maximum context length of 4K tokens. The used API of GPT-4 is gpt-4-0613 with a maximum context length of 8K tokens.

The analysis experiments with Code LLama and GPT models may suffer from data leakage issues. As mentioned in Appendix B, both CrossCodeEval and our ReccEval devote effort to collecting projects (e.g., released between 2023-01-01 to 2023-04-28 for ReccEval) that are not in pretraining corpora. However, Code Llama is trained on unknown datasets between January 2023 and July 2023, and the training data of GPT models is unknown and updated.

We set the temperature of these LMs as 0 to obtain deterministic results. The maximum generation length is set to 48 tokens, which is long enough to accomplish line completions. An exception is RG-1, which asks LMs to generate 100 tokens since RepoCoder requires sufficient content for further retrieval. We run StarCoder-15.5B, CodeGen-16.1B, and Code Llama-7B on an NVIDIA A800 with 80GB memory and run other LMs on an NVIDIA GeForce RTX 4090 with 24GB memory.

The instruction of GPT models is "You are a Python expert. Please complete the last line of the following Python code:". For RG-1, the instruction is "You are a Python expert. Please complete the following Python code:". We set line feed as the stop token of LM generation, except for RG-1 (still 100 tokens).

E Additional Evaluation

E.1 More Performance Comparison Results

Beyond the experimental results of the main paper, we show additional evaluation results of other CodeGen models in Tables 9 and 10. The additional results show consistent conclusions on performance comparisons in the main paper. Under the same architecture of the CodeGen-* models, the performance of all methods improves as the model parameters increase. Moreover, the improvement of DRACO for zero-shot code LMs increases as the model's capability grows. It indicates that

Methods	SantaCoder-1.1B		CodeGen25-7B		StarCoder-15.5B	
	CrossCodeEval	ReccEval	CrossCodeEval	ReccEval	CrossCodeEval	ReccEval
CCFinder-1	30	52	23	34	26	45
CCFinder-2	47	72	37	51	42	66
RG-1	**15**	**15**	**18**	**20**	**12**	**15**
RepoCoder	3,075	3,184	5,249	4,772	4,746	4,675
DRACO	35	42	32	36	60	77

Table 11: Prompt generation time (in milliseconds) of each method using SantaCoder, CodeGen25, and StarCoder models (cf. Table 5).

Methods	CodeGen-2.7B		CodeGen-6.1B		CodeGen-16.1B	
	CrossCodeEval	ReccEval	CrossCodeEval	ReccEval	CrossCodeEval	ReccEval
CCFinder-1	32	49	32	49	32	49
CCFinder-2	52	82	52	82	52	82
RG-1	**14**	**15**	**19**	**14**	**12**	**13**
RepoCoder	6,933	5,779	7,543	6,236	7,289	7,137
DRACO	40	44	40	44	40	44

Table 12: Prompt generation time (in milliseconds) of each method using other CodeGen models (cf. Table 5).

stronger LMs can better utilize the relevant background knowledge retrieved by DRACO.

E.2 More Efficiency Evaluation Results

We also record the time spent on indexing the repositories of CrossCodeEval and ReccEval, as shown in Table 14. It is an offline preprocessing in RAG, which indicates the time required to activate a method. CCFinder and DRACO build retrieval databases by statically parsing code files, which are independent of the used code LMs. RG-1 and RepoCoder need to tokenize the code snippets within a sliding window, which requires the tokenizers of used LMs. Note that the tokenizers of CodeGen-* models are the same. DRACO is 3–6 times faster than RepoCoder in preprocessing time. As the size of the repository increases, the preprocessing time grows linearly. Therefore, RG-1 and RepoCoder may suffer from scalability challenges.

The prompt generation time of each method using other code LMs is shown in Tables 11 and 12, which show consistent conclusions with the main paper. For the methods with one retrieval, only the tokenizers have a subtle effect on efficiency when different models are employed. As a result, the prompt generation time using different CodeGen-* models is the same for CCFinder, RG-1, as well as DRACO. RepoCoder relies on RG-1 to generate sufficient content for the second retrieval, where the efficiency mainly depends on the generation time of code LMs. In general, the generation efficiency of RepoCoder decreases as the model parameters increase. Its average prompt generation time is more than 3 seconds on the most efficient SantaCoder model, which far exceeds the time spent by other retrieval-augmented methods. Note that the architectures of code LMs also matter in efficiency, e.g., SantaCoder-1.1B is faster than CodeGen-350M. The A800 GPU used to run the StarCoder-15.5B and CodeGen-16.1B models is superior to the RTX 4090 GPU used for the other models, so these are not head-to-head comparisons for RepoCoder.

E.3 Effect of Other Import Statements

As described in Section 3.4, DRACO includes as many other `import` statements that are not directly relevant as possible. To analyze the effect of other `import` statements, we implement a variant of DRACO for comparison, which adds the entities from other local `import` statements only when the primary prompt is empty. The experimental results are shown in Table 13. DRACO consistently outperforms this variant, especially on StarCoder-15.5B, which has capability to understand longer code context.

F Case Study

F.1 Prompt Examples

We show the prompts generated by each method for the example unfinished code (see Figure 1). The prompts are excerpted for viewing the individual format, as shown in Figure 6. It can be observed that the prompts generated by DRACO look like natural code, which is in line with the training corpora of code LMs. The prediction result of each method using the CodeGen25-7B model is shown in Table 15, and only our DRACO generates the correct code line.

Settings		CodeGen-350M				SantaCoder-1.1B				CodeGen25-7B				StarCoder-15.5B			
		EM	ES	ID.EM	F1	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1
CrossCodeEval	DRACO	**13.02**	**61.30**	**20.53**	**49.04**	**20.64**	**67.04**	**29.83**	**57.37**	**24.99**	70.10	34.63	61.14	**34.67**	**75.83**	**45.63**	**69.93**
	Variant	12.87	61.20	20.45	48.76	20.45	67.03	**29.83**	57.36	24.77	**70.22**	**34.71**	**61.37**	33.55	75.58	44.50	69.33
ReccEval	DRACO	**22.12**	**60.41**	**29.73**	**46.09**	**30.26**	**66.90**	**39.08**	**55.43**	**36.46**	**70.76**	**44.67**	**60.40**	**46.49**	**76.80**	**55.98**	**70.32**
	Variant	21.34	60.01	29.05	45.57	29.36	66.36	38.04	54.67	35.51	70.18	43.82	59.52	44.47	75.67	53.92	68.89

Table 13: Performance comparison between DRACO and its variant (containing only the relevant `import` statements).

```
'''
# pyPhasesRecordloader.RecordSignal.RecordSignal
@classLogger
class RecordSignal:

# pyPhasesRecordloader.RecordSignal.RecordSignal.__init__
def __init__(self, targetFrequency=200, recordId=None):
        self.recordId = recordId
        self.signals: List[Signal] = []
        self.labelSignals = []
        self.signalNames = []
        self.targetFrequency = targetFrequency

# pyPhasesRecordloader.RecordSignal.RecordSignal.getSignalByName
def getSignalByName(self, name) -> Signal:
        index = self.getSignalIndexByName(name)
        return self.signals[index]
...
'''
```

(a) CCFinder-*.

```
# Here are some relevant code fragments from other files
  of the repo:
# --------------------------------------------------
# the below code fragment can be found in:
# pyPhasesRecordloader-0.3.12/pyPhasesRecordloader/RecordLoader.py
# --------------------------------------------------
#             signalTypeStr = self.signalTypeDict[signalName]
#         else:
#             self.logError("Signal '%s' had no type when
                  initilizing the RecordLoader" % str(signalName))
#             signalTypeStr = "unknown"
#         return signalTypeStr
    ...
# --------------------------------------------------
# the below code fragment can be found in:
# pyPhasesRecordloader-0.3.12/pyPhasesRecordloader/RecordSignal.py
...
```

(b) RepoCoder, same as RG-1.

```
'''
# pyPhasesRecordloader/Signal.py
class Signal:
    def __init__(
        self, name="Unknown", signal: np.ndarray = None,
        frequency=100, type=SignalType.UNKNOWN, typeStr="unknown"
    ) -> None:
        self.name = name
        self.signal = signal
        self.frequency = frequency
        self.type = type
        self.typeStr = typeStr

    def setSignalTypeFromTypeStr(self):
        if self.typeStr in signalTypeDict:
            self.type = signalTypeDict[self.typeStr]
        else:
            self.type = SignalType.UNKNOWN
    ...
'''
```

(c) Our DRACO.

```
'''
# pyPhasesRecordloader/Signal.py
class Signal:
    def __init__(
        self, name="Unknown", signal: np.ndarray = None,
        frequency=100, type=SignalType.UNKNOWN, typeStr="unknown"
    ) -> None:
        self.name
        self.signal
        self.frequency
        self.type
        self.typeStr

    def setSignalTypeFromTypeStr(self):
    def getFilterCoefficients(self, tansitionWidth=15.0,
                              cutOffHz=30.0, rippleDB=40.0):
    def bandpass(self, low, high, order=10):
    def lowpass(self, value, order=10):
    ...
'''
```

(d) DRACO with the *definition* scope.

Figure 6: Excerpts of example prompts generated by different methods.

Methods	Models	CrossCodeEval	ReccEval
CCFinder	All	74	68
RG-1 & RepoCoder	CodeGen	227	217
	SantaCoder	245	221
	CodeGen25	349	339
	StarCoder	211	186
DRACO	All	**54**	**61**

Table 14: Preprocessing time (in milliseconds) for the repositories in CrossCodeEval and ReccEval.

Methods	Predictions	ES
Zero-Shot	channel = newChannelName	24
CCFinder-1	type = Signal.getType(channelType)	53
CCFinder-2	type = Signal.getType(channelType)	53
RG-1	type = channelType	36
RepoCoder	signal = newSignal.signal.astype(channelType)	45
DRACO	setSignalTypeFromTypeStr()	100
Ground truth	setSignalTypeFromTypeStr()	-

Table 15: The example prediction of each method using the CodeGen25-7B model.

F.2 Study on Failed Cases

Despite the superior performance achieved by DRACO, there are still many failed cases. Based on our observations, they are caused by three major reasons: (i) The definitions of completion targets may be truncated. Even though the correct definition is retrieved through dataflow analysis, the completion target may be truncated due to the limited input length, e.g., target member function in a long class definition. (ii) The used code LMs may be not powerful enough, which has already been demonstrated in Section 5.5. It is crucial for code LMs to capture data dependency relations in the provided background knowledge, where the prompts are usually long Python code. (iii) Unclear code intent may lead to wrong generation, which is a common sore point of code completion. For example, completing the first line of a new function is uncertain even to a programmer.

In Listings 1, 2, and 3, we show an example of reasons (i) and (iii) from CrossCodeEval. The ground truth is `gen_begin_reuse(input_ids)`, while the prediction of DRACO with CodeGen25 is `in_beam_search = False`. Although the member function `gen_begin_reuse` is in the retrieved

background knowledge, it is truncated to be invisible to LMs. Moreover, the comment `# Start generation` is not clear enough for completion, where both `gen_begin` and `gen_begin_reuse` look like rational choices.

```
from model import ExLlama, ExLlamaCache,
    ExLlamaConfig
from lora import ExLlamaLora
from tokenizer import ExLlamaTokenizer
from generator import ExLlamaGenerator
import model_init

generator: ExLlamaGenerator

def cached_tokenize(text: str):
    ...

def begin_stream(...):
    global model, cache, config,
        generator, tokenizer

    # Settings
    max_stop_string = 2
    for ss in stop_strings:
        ...
    generator.settings = gen_settings

    # Start generation
    generator.
```

Listing 1: The unfinished code to be completed.

```
'''
# generator.py
class ExLlamaGenerator:
    class Settings:
        temperature = 0.95
        top_k = 40
        top_p = 0.65
        min_p = 0.0
        typical = 0.0
        token_repetition_penalty_max =
            1.15
        token_repetition_penalty_sustain
            = 256
        token_repetition_penalty_decay =
            128
        beams = 1
        beam_length = 1
    sequence: torch.Tensor or None
    sequence_actual: torch.Tensor or
        None
    settings: Settings
    beams: int or None
    max_beam_length: int
    in_beam_search: True
    disallowed_tokens: list[int] or None
    lora: ExLlamaLora or None
    def __init__(self, model, tokenizer,
        cache):
        self.model = model
        self.tokenizer = tokenizer
        self.cache = cache
        self.reset()
        self.model = model
        self.tokenizer = tokenizer
        self.cache = cache
    def reset(self):
        ...
    def make_rep_mask(self, penalty_max,
        sustain, decay):
        ...
    def batched_sample(self, logits,
        temperature, top_k, top_p, min_p
        , typical, num = 1):
        ...
    def sample_current(self, logits, num
        = 1):
        ...
    def sample(self, logits, temperature
        , top_k, top_p, min_p, typical,
        num = 1):
        ...
'''
```

Listing 2: The truncated background knowledge retrieved by DRACO.

```
'''
# generator.py
class ExLlamaGenerator:
    class Settings:
        ...
    sequence: torch.Tensor or None
    sequence_actual: torch.Tensor or
        None
    settings: Settings
    beams: int or None
    max_beam_length: int
    in_beam_search: True
    disallowed_tokens: list[int] or None
    lora: ExLlamaLora or None
    def __init__(self, model, tokenizer,
        cache):
        ...
    def reset(self):
        ...
    def make_rep_mask(self, penalty_max,
        sustain, decay):
        ...
    def batched_sample(self, logits,
        temperature, top_k, top_p, min_p
        , typical, num = 1):
        ...
    def sample_current(self, logits, num
        = 1):
        ...
    def sample(self, logits, temperature
        , top_k, top_p, min_p, typical,
        num = 1):
        ...
    def disallow_tokens(self, tokens):
        ...
    def gen_begin(self, in_tokens, mask
        = None):
        ...
    def gen_begin_empty(self):
        ...
    def gen_begin_reuse(self, in_tokens,
        mask = None):
        self.end_beam_search()
        ...
    def gen_feed_tokens(self, in_tokens,
        mask = None):
        ...
    def gen_accept_token(self, token):
        ...
    def gen_rewind(self, num_tokens):
```

```
        ...
    def gen_prune_right(self, tokens,
        mask = None):
        ...
    def gen_prune_to(self,
        min_tokens_to_keep, token_id,
        mask = None):
        ...
    def gen_prune_left(self, num_tokens,
         mask = None):
        ...
    def gen_num_tokens(self):
        ...
    def generate_simple(self, prompt,
        max_new_tokens = 128):
        ...
    def apply_rep_penalty(self, logits):
        ...
    def gen_single_token(self,
        constraints = None, mask = None)
        :
        ...
    class Beam:
        ...
    def begin_beam_search(self):
        ...
    def beam_search(self):
        ...
    def end_beam_search(self):
        if not self.in_beam_search:
            return
        self.in_beam_search = False
    def replace_last_token(self, token,
        seq = False):
        ...
    def sequence_ends_with(self, tokens)
        :
        ...
'''
```

Listing 3: The complete background knowledge retrieved by DRACO.

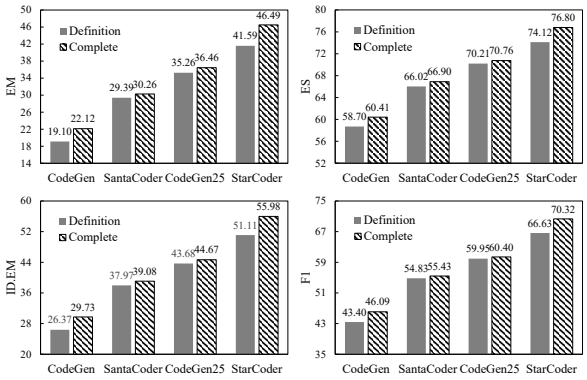

Figure 7: Performance comparison of two prompt scopes on the ReccEval dataset.

Methods	CodeGen-350M				SantaCoder-1.1B				CodeGen25-7B				StarCoder-15.5B			
	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1
DRACO	**22.12**	**60.41**	**29.73**	**46.09**	**30.26**	**66.90**	**39.08**	**55.43**	**36.46**	**70.76**	**44.67**	**60.40**	**46.49**	**76.80**	**55.98**	**70.32**
w/o cross_df	19.75	58.95	27.19	43.52	27.05	65.12	35.61	52.23	32.95	68.97	40.89	56.97	42.01	74.40	51.21	65.89
w/o intra_df	16.67	57.28	23.62	40.11	23.03	62.87	31.09	47.89	27.83	66.42	35.66	52.25	43.88	75.39	53.07	67.62
w/o dataflow	15.45	56.40	22.33	38.73	21.58	62.01	29.62	46.44	26.42	65.65	34.14	50.67	40.46	73.63	49.45	64.37

Table 16: Ablation study for dataflow analysis on the ReccEval dataset.

Methods	2K				4K				8K			
	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1
Zero-Shot	9.49	61.97	16.44	47.36	9.76	62.17	16.70	47.51	9.61	62.14	16.59	47.51
CCFinder-1	17.11	66.28	26.02	55.34	18.12	66.99	27.17	56.54	17.79	67.28	27.05	56.65
RG-1	17.82	67.46	27.43	56.51	21.88	69.60	31.44	59.65	25.10	71.84	36.06	63.27
RepoCoder	19.10	**68.96**	29.83	58.77	**24.24**	**71.29**	**35.01**	**62.39**	**28.29**	**73.51**	**39.44**	**65.24**
DRACO	**21.35**	68.78	**30.66**	**59.08**	20.94	68.65	30.09	58.68	20.08	68.42	29.12	58.32

Table 17: Performance comparison of the Code Llama-7B model (with 2K, 4K, 8K maximum input lengths) on the CrossCodeEval dataset.

Methods	2K				4K				8K			
	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1
Zero-Shot	13.85	58.82	20.60	38.01	13.93	58.96	20.74	38.14	13.93	59.04	20.74	38.26
CCFinder-1	26.51	65.92	34.62	51.08	28.06	66.95	36.33	52.68	28.01	67.21	36.12	52.74
RG-1	30.55	67.90	37.78	52.66	36.64	71.15	44.44	58.22	41.65	73.95	49.64	63.05
RepoCoder	**34.10**	**69.65**	**41.50**	56.17	**40.55**	**72.97**	**48.38**	**61.49**	**44.76**	**75.31**	**52.28**	**65.43**
DRACO	31.54	68.87	40.02	**56.26**	33.25	69.68	42.02	58.06	30.04	68.06	38.26	54.58

Table 18: Performance comparison of the Code Llama-7B model (with 2K, 4K, 8K maximum input lengths) on the ReccEval dataset.

Methods	2K				4K				8K			
	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1
Zero-Shot	8.71	61.95	16.02	47.51	8.74	62.07	16.06	47.63	8.71	62.08	16.02	47.58
CCFinder-1	22.51	69.15	31.82	59.77	26.08	71.29	35.83	62.80	27.99	72.59	38.24	64.46
RG-1	18.31	68.25	28.37	57.27	21.69	70.50	31.97	60.53	26.27	72.70	37.00	64.04
RepoCoder	20.26	69.84	30.51	59.31	24.77	72.69	36.25	63.57	29.12	74.56	40.83	66.81
DRACO	**28.78**	**72.39**	**38.72**	**64.90**	**32.80**	**74.89**	**43.49**	**68.42**	**34.67**	**75.83**	**45.63**	**69.93**

Table 19: Performance comparison of the StarCoder-15.5B model (with 2K, 4K, 8K maximum input lengths) on the CrossCodeEval dataset.

Methods	2K				4K				8K			
	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1	EM	ES	ID.EM	F1
Zero-Shot	12.55	58.65	19.72	37.89	12.85	58.80	20.03	38.14	12.77	58.84	20.03	38.12
CCFinder-1	30.69	68.35	39.13	55.32	36.19	71.13	44.70	60.21	39.33	73.05	48.18	63.49
RG-1	30.78	68.33	38.23	53.34	36.88	71.64	44.78	59.06	42.67	74.64	51.11	64.64
RepoCoder	34.62	70.39	42.41	56.94	40.35	73.32	48.26	62.37	46.26	76.44	54.47	67.59
DRACO	**39.61**	**73.32**	**48.85**	**64.44**	**44.03**	**75.42**	**53.34**	**68.26**	**46.49**	**76.80**	**55.98**	**70.32**

Table 20: Performance comparison of the StarCoder-15.5B model (with 2K, 4K, 8K maximum input lengths) on the ReccEval dataset.