CoCoMIC: Code Completion By Jointly Modeling In-file and Cross-file Context

Yangruibo Ding[1,*,†], **Zijian Wang**[2,*,‡] **Wasi Uddin Ahmad**[2,*]
Murali Krishna Ramanathan[2] **Ramesh Nallapati**[2] **Parminder Bhatia**[2]
Dan Roth[2] **Bing Xiang**[2]

[1]Columbia University [2]AWS AI Labs
`yrbding@cs.columbia.edu` `{zijwan,wuahmad}@amazon.com`

Abstract

While pre-trained language models (LM) for code have achieved great success in code completion, they generate code conditioned only on the contents within the file, *i.e., in-file context*, but ignore the rich semantics in other files within the same project, *i.e.,* project-level *cross-file context*, a critical source of information that is especially useful in modern modular software development. Such overlooking constrains code LMs' capacity in code completion, leading to unexpected behaviors such as generating hallucinated class member functions or function calls with unexpected arguments. In this work, we propose CoCoMIC, a novel framework that jointly learns the in-file and cross-file context on top of code LMs. To empower CoCoMIC, we develop CCFinder, a static-analysis-based tool that locates and retrieves the most relevant project-level cross-file context for code completion. CoCoMIC successfully improves the existing code LM with a 33.94% relative increase in exact match and 28.69% in identifier matching for code completion when the cross-file context is provided. Finally, we perform a series of ablation studies and share valuable insights for future research on integrating cross-file context into code LMs.

Keywords: Code Completion, Code Generation, Repository-level Code Completion

1. Introduction

In recent years, language models for source code like Codex (Chen et al., 2021) and CodeGen (Nijkamp et al., 2023) have shown promising performance in code completion tasks and have great potential to improve developer productivity (Barke et al., 2023). These code LMs are typically trained with causal language modeling loss and complete the code conditioning on the previous code tokens in the same file, which we refer to as *in-file context*.

Modular programming (Parnas, 1972; Parnas et al., 1985; Sullivan et al., 2001) is a software design strategy that divides the complex software functionality into several independent, interchangeable components (*e.g.,* files, classes, and functions), such that each component implements only one aspect of the desired functionality and consequently becomes easily reusable and testable. It has already been a well-adapted paradigm in modern software development and maintenance. Developing under the modular programming paradigm requires knowledge from the current file and the whole project, to which we refer as *cross-file context*. As shown in Figure 1, the *cross-file context* is critical for code completion: the CodeGen Python model (Nijkamp et al., 2023) with 2 billion parameters fails to generate the correct code since it only considers *in-file context* and lacks visibility to various crucial references for code completion, *e.g.,* member functions of imported classes and arguments of imported functions.

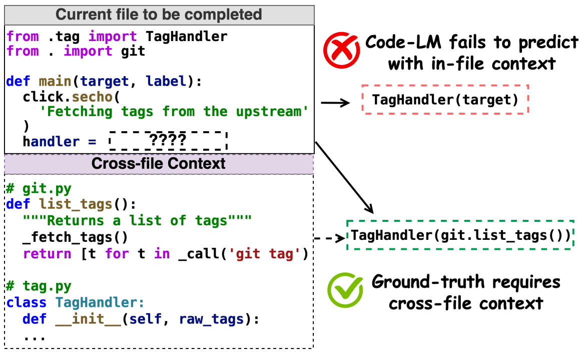

Figure 1: CodeGen-2B-mono fails to complete a Python program correctly as *in-file context* does not provide sufficient information. The model needs to know that `TagHandler` takes an argument `raw_tags`, which could be obtained through the function `list_tags` of `git`. Generating the correct code requires the presence of class and function definitions as part of the context, which cannot be derived from the current file alone.

In this work, we argue that code LMs should generate code conditioned jointly on *in-file context* and *cross-file context*. However, there are challenges in developing such models. First, the project defines its individual and complex hierarchy and could be of varied sizes. Thus, given a piece of code, it is critical yet challenging to efficiently identify the most relevant and useful cross-file context. Second, we must carefully design a framework for aggregating

LREC-COLING 2024, pages 3433–3445
20-25 May, 2024. © 2024 ELRA Language Resource Association: CC BY-NC 4.0

the information from the in-file and cross-file context. Naïvely concatenating code from in-file and cross-file context is not feasible for two reasons. First, they represent distinct types of contextual information, as the former presents the local dependencies and human intentions (*e.g.,* code comments) for code completion, while the latter compensates for the project-level dependencies that do not exist in the surrounding lines. Thus, the model should *not* always treat them equally. Second, the model's input length is limited, so concatenating all contexts as input will exceed its context length.

Besides, unlike third-party packages, which are mostly available in the pre-training dataset of code LMs, the project-level context is likely to be private to the model, given its under-development nature. As illustrated in Figure 1, code LMs demonstrate diminished performance and hallucination when code completion necessitates cross-file dependencies from the ongoing private project.

To address the aforementioned challenges, we propose CoCoMIC, a novel framework that jointly learns in-file and cross-file context to improve code completion. To automatically retrieve the most relevant cross-file context, we further build a static-analysis-based cross-file context finder, CCFinder, that effectively fulfills this task.

Cross-file Context Finder We design and implement CCFinder, a static code analysis tool, to retrieve the most relevant cross-file context for code completion. CCFinder parses the project hierarchy and code components to extract project information. CCFinder further builds a project context graph to represent the details of each component (*i.e.,* entity) and the interactions among them (*i.e.,* relation). When an incomplete program requests completion, the tool will first analyze its `import` statements and pinpoint the related entities from the built context graph. Then, the tool will retrieve the neighbors of the pinpointed entities from the graph as the cross-file context of the current file.

Jointly Modeling In-file and Cross-file Context We propose CoCoMIC, a novel framework built on top of existing code LMs with joint attention to in-file and retrieved cross-file context. We realize this in two steps: First, the model will compress cross-file context and build its representations. Second, when generating code completion, the model will attend to both the compressed cross-file context and the concrete in-file context.

We evaluate the effectiveness of CCFinder and CoCoMIC on a code completion dataset we built from the Python Package Index (PyPI), a repository of open-source Python projects. We show that CCFinder can retrieve 27.07% more relevant context for code completion than in-file context. By integrating the retrieved context from CCFinder, CoCoMIC improves the backbone pre-trained code LM, CodeGen (Nijkamp et al., 2023), by 33.94% in exact match and 28.69% identifier matches relatively. Our main contributions are as follows.

1. Our work sheds light on the importance of project-level cross-file context, a critical yet overlooked resource in the era of language models for code completion.
2. We present CoCoMIC, a novel framework built on top of code LMs that jointly learns in-file and cross-file context to enhance code completion (§4). To empower CoCoMIC, we develop CCFinder, an effective static-analysis-based tool that collects the most relevant cross-file context to be integrated into CoCoMIC (§3).[1]
3. We show that CoCoMIC with cross-file context from CCFinder significantly outperforms fine-tuned baselines by up to $+33.94\%$ in exact match. We additionally conduct extensive ablation studies to show the contribution of different components (§5 & 6).

2. Preliminaries

For the convenience of discussion, we define concepts that will be used throughout the paper.

Project Entities Project entities are code components that constitute the skeleton of software projects; developers frequently import and reuse these entities as cross-file context. We focus on four types of entities: *file, function, class, and global variable*. In particular, *file* contains the file name and file docstring; *class* contains the class signature, docstring, and attributes; *function* contains function signature, docstring, and body; *global variable* contains the variable name and its value.

Entity Relations Entity relations represent the interactions among project entities. We consider two categories of relations: *intra-file* and *inter-file*. Intra-file relations describe the in-file code hierarchies pre-defined by the programming language grammar. For example, a class is at the first level of the hierarchy while its member functions are at the second level. Inter-file relations define the file-to-file dependencies. Under each category, we further define several types of relations.

Locale We define *locale* as the entity's relative code location within the software project. For example, the locale of class entities is defined as `file_name.class_name`. The locale is assigned a unique name according to the specific location of a project entity, so we maintain the one-to-one mapping between each entity and its locale. The locale benefits CoCoMIC in two ways: (1) when we construct cross-file context, the locale efficiently maps the relative path of a code snippet to its project entity CCFinder builds (§3.2), and (2)

[1]We will release our code at https://github.com/amazon-science/cocomic

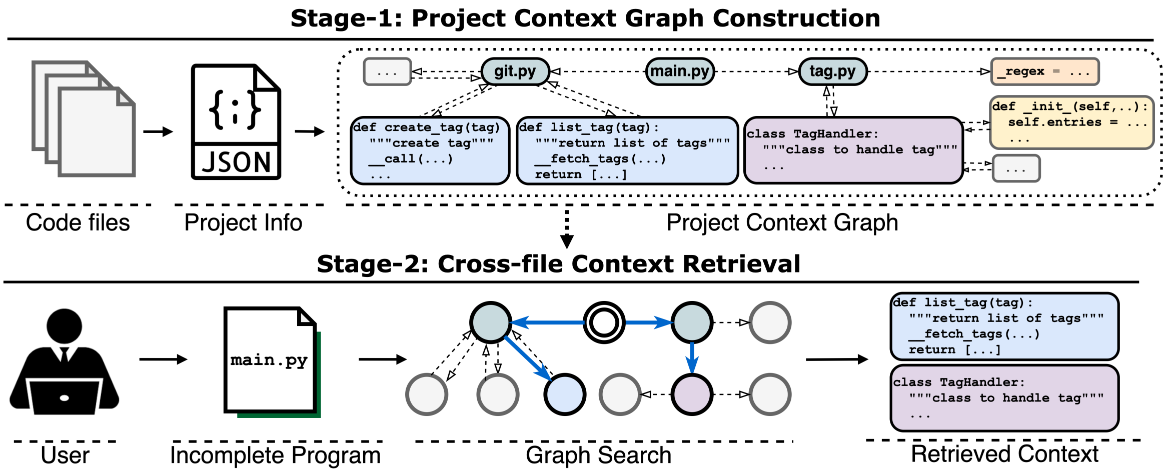

Figure 2: Overview of CCFinder. First, CCFinder builds the project context graph, including the bird's-eye view of the whole project and the code details of each module. Then, given the incomplete program, CCFinder retrieves a set of the most relevant project entities as cross-file context from the graph.

it indicates hierarchical relations among project entities and helps model with code completion (§6.4).

In-file & Cross-file Context For an incomplete source file $\mathcal{S}$, we define two types of context: *in-file* and *cross-file*. In-file context represents code snippets included in the current file, *i.e.,* code tokens before the predicting position. Cross-file context $\mathcal{C}$ represents the relevant code information (*e.g.,* classes, functions) from the same project that is out of but imported by the current file. Concretely, *cross-file context* refers to a collection of relevant project entities that might assist with the missing code prediction but are not in $\mathcal{S}$.

3. Cross-file Context Finder

Software projects typically have complex structures (Parnas et al., 1985) representing the dependencies among distinct code components. To retrieve the most relevant cross-file context as the code LM's additional reference, we need a tool with three main characteristics. First, it should be able to navigate the project structure to identify the file and module dependencies. Second, it can zoom into the dependencies and extract detailed code components. Third, given a code sample, the process of cross-file context retrieval should be stable and automated for large-scale training and inference. Unfortunately, a single off-the-shelf tool could not meet all three requirements. For example, module dependency analysis tools[2,3] can only provide the module interactions while missing the hierarchical details inside each module and cannot directly output the concrete code. Therefore, we develop a new static-analysis-based tool, CCFinder, to automatically collect the most relevant cross-file context that will be integrated into code LMs (§4).

[2] https://github.com/google/importlab
[3] https://github.com/thebjorn/pydeps

CCFinder's overall workflow is shown in Figure 2. It has two main steps: (1) Analyze the program dependencies to build a bird's-eye view of the whole project and parse the source code to extract code details of each module. With these, CCFinder builds the *project context graph*: graph nodes represent code components that constitute the project's backbone, and edges indicate the relations among components. (2) Given an incomplete program, the tool retrieves the most relevant cross-file context from the built graph. In this work, we focus on Python as the proof-of-concept to showcase our main arguments. However, CCFinder's conceptual design is extensible to other languages.

3.1. Project Context Graph

CCFinder parses the project structure and corresponding source files to identify the project entities and entity relations. Then, CCFinder uses entities and entity relations to build graph nodes and directed edges, respectively. The context graph is built top-down. First, we create a root node for the project and connect it with all file nodes. Second, each file node will build its own sub-graph, wrapping code components within the file, and also build connections with other files that it depends on, *i.e.,* it imports code from these files. Third, nodes will link to others within the file-level sub-graph based on the dependencies or scope. For example, a class node will have edges to its member functions.

Formally, CCFinder builds the multi-relational, directed context graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ for the project, where $\mathcal{V}$ is the set of nodes representing code components, and $\mathcal{E}$ is the set of edges that indicate the interactions among code components.

3.2. Cross-file Context Retrieval

In the project context graph, the closer a graph neighbor is to a specific code snippet (*i.e.,* entity), the more relevant that neighbor is. For example, the detailed information of an imported file entity, such as its defined functions or classes, should be only 1 or 2 hops away. Therefore, we first analyze the `import` statements of an incomplete program to pinpoint entities related to its cross-file dependencies. Then we retrieve their neighbors within 2 hops using the depth-first graph search. We justify that 2-hop neighbors include comprehensive context for code completion in Section 6.5. The set of retrieved nodes is used as cross-file context, which will maintain their relative order according to the original source file.

4. The CoCoMIC Framework

Figure 3 presents the high-level overview of the CoCoMIC framework. CoCoMIC uses an autoregressive LM to encode 1) in-file code snippet and 2) retrieved cross-file context, then predicts the next code token conditioning on both. CoCoMIC is model-agnostic and we use CodeGen, one of the most popular code LMs, to demonstrate later (§5).

4.1. Input Representation

As shown in Figure 3, the model input includes two parts: source code sample $\mathcal{S}$ and its cross-file context $\mathcal{C}$. Specifically, the source code sample $\mathcal{S}$ consists of a sequence of tokens $x_1, ..., x_T$, where x_t is a code token and T is the length of $\mathcal{S}$; the cross-file context, as introduced in §3, is a list of entities, $\mathcal{C} = (c_1, ..., c_n)$, retrieved from the project context graph. Each entity, c_i, is a short piece of code sequence describing the details of that entity, *i.e.,* $c_i = (locale_i, w_i^1, ..., w_i^m, \texttt{[SUM]})$, where w_i^j is a code token within the entity, $locale_i$ is the locale (§2) of c_i, and `[SUM]` is a special token.

Representing Entity Relations with Locales As introduced in §2, each project entity is paired with a locale that indicates its hierarchical relationship. We explore the benefits of prepending locales to provide entities with such relational hints (§6.4). Specifically, for each cross-file entity, we prepend its locale to its code text as a comment, followed by a new line character: for the example in Figure 3, the retrieved entity `def list_tags()` will be prepended with `#git.list_tags\n`.

Better Entity Representation with `[SUM]` We append a special token `[SUM]` to entity descriptions. We expect `[SUM]` token to learn the summarization of the entity since the causal attention (Radford et al., 2019; Brown et al., 2020) allows it to attend to all the previous tokens describing the entity. When completing code, the model will attend to the representations of the `[SUM]` tokens for each cross-file entity. We compare it with mean pooling in §6.3 and show that `[SUM]` works better.

4.2. Encoding Cross-file Context

The computational cost of Transformers increases exponentially *w.r.t.* the input length, so it is impractical to prepend all the retrieved entities as plain text, as they typically contain thousands of tokens. Also, only a few keywords in an entity (e.g., identifiers) play an important role in assisting code completion. Thus, CoCoMIC encodes each entity into a single token to balance the space limitation and the information needed.

$$h_{c_i} = f_\theta(c_i) \in \mathbb{R}^{d_h}; H_\mathcal{C} = (h_{c_1}, ..., h_{c_n}) \in \mathbb{R}^{n \times d_h}$$

Specifically, for each entity c_i, the model f_θ will encode its code sequence into one representation $h_{c_i} \in \mathbb{R}^{d_h}$, where d_h is the hidden dimension. Then, CoCoMIC takes the hidden state of the last token, `[SUM]`, as the entity representation. Finally, the model will output a list of entity embeddings, $H_\mathcal{C}$, representing the retrieved cross-file context.

4.3. In-file and Cross-file Context

After getting representations of cross-file context, CoCoMIC continues to encode the in-file context and train the model to learn both contexts jointly.

In-file Context CoCoMIC utilizes the causal language model setting to support the code completion task, where each token will consider its former texts as in-file context. Specifically, the in-file context of source code $\mathcal{S}$, at time step t, will be $s_t = (x_1, ..., x_{t-1})$. We pass these tokens through the model and get the embeddings of each token to construct the representation of the in-file context.

$$H_\mathcal{S}(t) = f_\theta(s_t) = f_\theta(x_1, ..., x_{t-1}) \in \mathbb{R}^{(t-1) \times d_h}$$

Joint attention to In-file and Cross-file Context Different layers of a Transformer model have been shown to capture different language components (*e.g.,* lower layers learn language syntax or grammar while upper layers capture language semantics (Jawahar et al., 2019)). We hypothesize that both in-file and cross-file contexts contribute to forming the understanding of language components. Therefore, we fuse the in-file and cross-file context at each Transformer layer so that generating the next token's hidden state will always depend on both contexts. At each time step t, for the l-th layer, we first compute the keys and values for cross-file and in-file context, using their $(l-1)$-th hidden states.

$$K_\mathcal{C} = H_\mathcal{C}^{[l-1]} \mathbf{W}^K, V_\mathcal{C} = H_\mathcal{C}^{[l-1]} \mathbf{W}^V$$
$$K_\mathcal{S}(t) = H_\mathcal{S}(t)^{[l-1]} \mathbf{W}^K, V_\mathcal{S}(t) = H_\mathcal{S}(t)^{[l-1]} \mathbf{W}^V$$

Then, we concatenate the keys and values from both contexts so that, at time step t, the generating

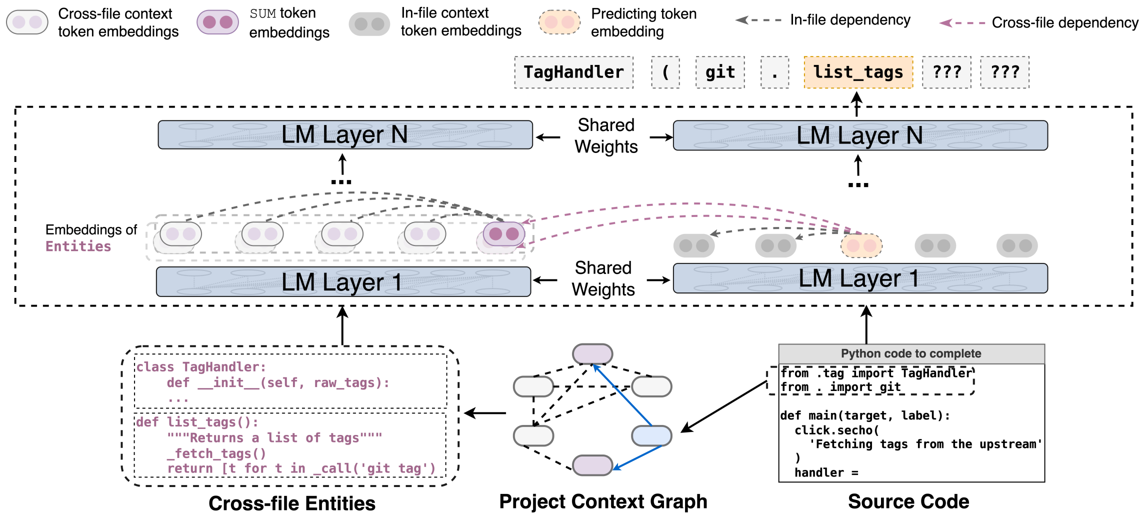

Figure 3: The CoCoMIC framework. **Bottom**: Given incomplete code, CoCoMIC leverages CCFinder to identify the corresponding entities in the project context graph (§3.1) and retrieve their k-hop neighbors as cross-file entities (§3.2). **Up**: CoCoMIC first generates representations for cross-file entities using the appended `[SUM]` token (§4.2). Then it completes the current code by jointly attending to in-file and cross-file context (§4.3).

token can jointly attend them.

$$K(t) = K_{\mathcal{C}}||K_{\mathcal{S}}(t), V(t) = V_{\mathcal{C}}||V_{\mathcal{S}}(t),$$
$$Q(t) = f_\theta(x_t)^{[l-1]}\mathbf{W}^Q,$$
$$Attn(t) = \texttt{softmax}(\frac{Q(t)K(t)^\top}{\sqrt{d_K}})V(t),$$

where $||$ indicates the concatenation of vectors.

5. Experiment Setup

5.1. Data

Our data stem from the Python Package Index (PyPI). We collect permissively licensed projects and filter out those with $\leq$5 python files or $\geq$5k nodes in project context graph, ending up with 60,891 projects. Then, we divide the dataset into 80%/10%/10% train, validation, and test sets. We notice that popular packages, such as `numpy`, are used as dependencies by many packages and will cause potential information leakage if `numpy` is part of the test set. Thus, we only include projects that were not used as dependencies by any training projects in the test set. We create prompts by cutting the source file at the location where completion requires cross-file context. We present the sequence length statistics in Table 1. For cross-file context, we concatenate the text of all retrieved entities as a sequence and count the length.

Figure 1 shows an example prompt we create: it requires the details of `TagHandler` and `git` to complete the code accurately. In this work, we consider statement-level code completion, so the ground truth of the test sample is built accordingly. For the convenience of studying the model's prediction on local APIs (*i.e.,* APIs defined within the project), we further filter out the samples that either can not be parsed by the AST parser or do not include local API calls in the target statement (to be completed). Finally, we ended up with the 6,888 held-out prompts for evaluation.

	Mean	Max	Median	Min
Prompts	1,354	32,599	758	7
Cross-file Context	4,485	186,339	1,928	22

Table 1: Number of tokens using CodeGen's tokenizer of prompts and cross-file context of the test set.

5.2. Implementation Details

Cross-file Context CCFinder uses tree-sitter[4] to parse source code files. Tree-sitter is a widely used source code parser that generates the abstract syntax tree (AST) given a program. CCFinder will traverse the AST to extract information as described in §3. Then, CCFinder analyzes the `import` statements on top of import-dep[5] to build the project context graph. In this work, we retrieve 2-hop neighbors with at max 128 project entities as cross-file context, and each entity contains up to 128 tokens. These thresholds are data-driven to ensure the model input covers most of the relevant cross-file context.

Model The backbone of CoCoMIC is CodeGen (Nijkamp et al., 2023) and we use CodeGen-350M-Mono for all experiments. In all settings, we fine-

[4] https://tree-sitter.github.io
[5] https://pypi.org/project/import-deps

tune the model for 5 epochs with a max sequence length of 2,048 tokens and a learning rate of 5e-5 with 5% warm-up steps, then cosine annealing.

Our code is based on Transformers (Wolf et al., 2020). We train our models on a machine with 8 Nvidia A100s. Each job takes around 50 hours (i.e., 400 GPU hours) to train all models.

5.3. Baselines & Evaluation Metrics

CodeGen We consider two variations of the vanilla CodeGen model with the in-file context only: (1) zero-shot, where we directly evaluate the pretrained CodeGen model on our test dataset, and (2) finetuned, where we finetune CodeGen on our dataset first and then evaluate.

CodeGen w/ Cross-file Context We also consider a prompting baseline where we prepend the cross-file context to the input sequence and finetune. Similar to the configuration of CoCoMIC, we reserve the first 128 tokens of the input for the code tokens from the cross-file context and use the rest tokens for the in-file context.

Evaluation Metrics We compute exact match (EM) and BLEU-4 (Papineni et al., 2002) to assess the accuracy of the generated code. While code match indicates the overall correctness of code completion, we want to zoom into the cases where cross-file context could most contribute, which is API usage. Therefore, we measure the identifier match to evaluate whether cross-file context improves the model's ability to predict the right APIs. To this end, we extract the identifiers from the model prediction and the ground truth, resulting in two ordered lists of identifiers. Then, we compare them and report the identifier prediction accuracy in terms of exact match, precision, and recall.

Besides, we compute the perplexity of all the tokens on the test set to study whether adding cross-file context degrades performance when the cross-file context is not explicitly required.

6. Results and Analysis

6.1. Main Results

We present the results in Table 2. CoCoMIC outperforms all baselines on all metrics with a clear margin, demonstrating the effectiveness of our proposed framework. We notice that when the cross-file context is prepended as a plain text prompt, CodeGen outperforms the other two baselines without cross-file context. However, limited by the maximum input length, it can only include a very limited amount of cross-file context, which significantly restricts its capacity. In contrast, CoCoMIC encodes the code sequence of an entity into one single token, enabling the model to incorporate more cross-file context while saving the input length.

Besides, we see no degradation when the cross-file context is not explicitly required. We calculate the perplexity of all tokens in the test samples, regardless of whether they require cross-file context. We see that CoCoMIC achieves the lowest perplexity, indicating cross-file context in CoCoMIC is generally beneficial for code completion.

6.2. Effectiveness of CCFinder

The objective of CCFinder is to locate and retrieve relevant code context from other source files in the project. Identifiers (e.g., function names and parameters) are presumably one of the most critical API information. Therefore, we study the effectiveness of CCFinder by assessing whether their retrieved-context increases recall of the identifiers that appear in the ground truth. We hypothesize that the inclusion of identifiers needed to complete a code is likely to benefit CoCoMIC.

Table 3 shows that the in-file context covers (recall) 75.19% identifiers that appear in the ground truth. In comparison, prompts augmented with retrieved cross-file identifiers bring up identifier recall to 95.55%. This indicates that CCFinder can retrieve most of the cross-file context that can help LM complete the input code. Note that while CCFinder increases identifier recall by 27.07%, Table 2 shows only an 8.97% improvement in identifier recall. This indicates that more intelligent prompting techniques or training better LMs to use cross-file context can lead to better performances. Further, Table 4 shows that random entities from the same project do not provide useful information since they are not necessarily related to the input code, and 2-hop retrieval outperforms 1-hop retrieval. These verify that CCFinder retrieves relevant cross-file context and thus helps CoCoMIC.

6.3. `[SUM]` Token Representing Entities

We append a special token `[SUM]` to cross-file context to summarize their information (Figure 3). Now, we study the importance of the `[SUM]` token for a better representation of cross-file context. As a comparison, we apply the widely-used mean pooling that takes the mean over every cross-file token's embedding as the cross-file representation. We train a CoCoMIC model with mean pooling and keep the rest of the settings the same. The result is in Table 5: our proposed `[SUM]` token effectively summarizes cross-file context and significantly outperforms the mean pooling strategy.

6.4. Impact of Locales in CoCoMIC

As introduced in §4.1, we prepend locales as relational hints for better entity representations. We study the effectiveness of such relational signals.

Model	Finetuned	Cross-file Entities	Code Match		ID Match			PPL (↓)
			EM	BLEU-4	EM	Prec.	Rec.	
CodeGen	✗	✗	14.56	33.12	22.91	47.74	50.75	2.88
+ Finetune	✓	✗	15.97	35.11	24.29	50.46	53.07	2.87
+ Cross-file context	✓	✓	17.00	36.34	25.80	48.91	54.76	2.77
CoCoMIC (Ours)	✓	✓	**21.39**	**41.65**	**31.26**	**55.45**	**57.83**	**2.69**

Table 2: Performance of CoCoMIC compared with baselines. We show that using the text prompt for cross-file entities (row 3) helps marginally compared to the in-file-only baseline (row 2). On the contrary, CoCoMIC with cross-file context (row 4) improves the performance by a large margin (+33.94% Code Match EM and +28.69% ID Match EM) compared to the in-file only baseline. In addition, we show that there is no degradation in perplexity (PPL) when evaluating all the tokens in the test set where the cross-file context is not always required, suggesting that adding cross-file context helps in general.

Code Context Type	ID Recall (%)
In-file context	75.19
In-file + Cross-file context	**95.55**

Table 3: CCFINDER retrieves 27.07% more identifiers when compared to only in-file contexts.

Entities From	Code Match		ID Match		
	EM	BLEU-4	EM	Prec.	Rec.
Random	15.68	35.23	24.07	49.75	52.69
CCFINDER (1-hop)	18.47	38.09	28.14	53.20	55.63
CCFINDER (2-hop)	**21.39**	**41.65**	**31.26**	**55.45**	**57.83**

Table 4: Entities retrieved from CCFINDER are more useful than random entities, and 2-hop retrieval help achieve better performance.

CoCoMIC	Code Match		ID Match		
	EM	BLEU-4	EM	Prec.	Rec.
Mean pooling	16.78	36.02	25.01	50.50	52.61
`[SUM]`	**21.39**	**41.65**	**31.26**	**55.45**	**57.83**

Table 5: `[SUM]` token representing cross-file context significantly outperforms mean pooling.

As a comparison, we further study multi-task learning that encourages embedding relational information into entity representations.

Multi-task w/ Edge Prediction We use multi-task learning (MTL) to encode cross-file relations. Specifically, we train the model with an auxiliary edge prediction task among cross-file entities. We take representations of two cross-file entities generated by the LM layers and ask the model to predict what edge type connects them.

Results Table 6 presents the results. While MTL achieves 97.2% accuracy in the auxiliary edge prediction task, it hardly improves CoCoMIC in code completion. Such a gap suggests that even if MTL fulfills the expectation of embedding edge information, this information is not directly useful for code completion. In contrast, adding locales consistently improves CoCoMIC across all metrics. We hypothesize that this is due to locales providing an exact and direct signal as text (e.g., `class_name.method_name`). Thus the model could use them as *short-cut* in code completion.

CoCoMIC	Code Match		ID Match		
	EM	BLEU-4	EM	Prec.	Rec.
No Relations	20.27	40.62	30.02	55.44	57.46
MTL	20.01	40.00	29.53	55.51	56.68
Locale	**21.39**	**41.65**	**31.26**	55.45	57.83
Locale + MTL	21.25	41.44	31.05	**55.83**	**58.03**

Table 6: Locales improve performance while learning cross-file relations with multi-task learning only provides CoCoMIC marginal improvement.

6.5. k-hop Retrieval

As we see from Table 4, $k = 1$ underperforms compared to $k = 2$. This is because $k = 1$ fetches less comprehensive context. For example, with the import statement `import FileA as A`, we can access class `X`'s static member function `Y` as: `A.classX.funcY` through 2-hop retrieval, whereas 1-hop retrieval will not fetch. In fact, 1-hop retrieval won't fetch any class member function if only the file is imported, which frequently happens in Python. Given the great coverage of $k = 2$ (Table 3) and given we found too many unrelated entities were retrieved if we use $k > 2$, we decided to use $k = 2$ throughout the work.

6.6. Re-ranking Cross-file Entities

The cross-file entities are organized and presented to CoCoMIC following their import order in the proposed design (§3.2). We also explore the effects of re-ranking cross-file entities according to their relevance to the prompt. Specifically, we use the Jaccard index-based (Jaccard, 1912) ranking and use the last 10 lines of code in the prompt as the query to re-rank all the entities obtained by CCFINDER.

	Code Match		ID Match		
	EM	BLEU-4	EM	Prec.	Rec.
CodeGen + Ft.	17.00	36.34	25.80	48.91	54.76
+ re-ranked ent.	17.76	36.98	26.31	48.27	55.87
CoCoMIC	21.39	41.65	31.26	55.45	57.83
+ re-ranked ent.	21.62	41.89	31.69	55.96	58.19

Table 7: Re-ranking cross-file entities marginally improves the performance of CoCoMIC and CodeGen model finetuned with cross-file context.

	Code Match		ID Match		
	EM	BLEU-4	EM	Prec.	Rec.
CodeGen + Ft.					
+ CFC (full)	17.00	36.34	25.80	48.91	54.76
+ CFC (simp.)	17.49	37.57	26.76	51.71	54.88
CoCoMIC	**21.39**	**41.65**	**31.26**	**55.45**	**57.83**

Table 8: CoCoMIC significantly outperforms all baselines even when more cross-file context (CFC) is included in the prompt for baselines.

The detailed results are shown in Table 7. Re-ranking cross-file entities does not significantly improve the CoCoMIC's performance. The improvement is only marginal due to (1) CoCoMIC efficiently encodes sufficient (up to 128 cross-file entities) cross-file context, so re-ranking could not bring more information, and (2) CoCoMIC's cross-context attention could make use of both in-file and cross-file context flexibly, so the input order of cross-file entities does not matter much. The baseline model, CodeGen finetuned with cross-file context, reports slightly more improvement when the cross-file entities are re-ranked. This is because the baseline model takes plain text as cross-file context, and a large portion of such information is truncated due to the limited input length, and thus prioritizing the most relevant entities to the prompt brings more useful information to the front and is included by the model input. However, the performance of the re-ranked and finetuned baseline is still far behind CoCoMIC, highlighting CoCoMIC is effective in modeling both in-file and cross-file context.

6.7. Additional Baseline Variants

In addition to the CodeGen w/ Cross-file Context baseline (§5.3), which uses the same cross-file context tokens as in CoCoMIC, we experimented with a simplified setting that only takes the locales and the signature prototypes (name, arguments, and default return types, if present) to fit in more cross-file context within the input length budget.

From Table 8, we see the performance only improves marginally when using simplified cross-file context, and it still underperforms CoCoMIC significantly. This suggests that baseline models have substantial limitations of sequence lengths that the performance is subpar even if we simplify the cross-file context, while CoCoMIC is capable of compressing up to 16,384 (=128x128) tokens of cross-file context into only 128 vectors, making cross-file context readily available for the model to use.

7. Related Work

In the last couple of years, a significant effort has been made to pretrain Transformer language models using unlabeled source code (Feng et al., 2020; Ahmad et al., 2021; Wang et al., 2021b; Guo et al., 2022; Ding et al., 2022b) to facilitate software engineering applications (Husain et al., 2019; Iyer et al., 2018; Tufano et al., 2019; Zhou et al., 2019). Among these efforts, developing code generation models is noteworthy (Chen et al., 2021; Xu et al., 2022; Wang and Komatsuzaki, 2021; Black et al., 2021a, 2022; Nijkamp et al., 2023; Fried et al., 2023; Li et al., 2022). Since most of these models are autoregressive language models, they can be directly used in code completion - given a code snippet as a prompt, generate the next N tokens. Until recently, existing works in the literature use code snippets from the current file (where the user is writing code) to prompt the code generation models.

While the use of in-file or class context is rigorously studied for software engineering applications in the literature, the use of cross-file context is relatively under-explored in code completion backed by code LMs. Earlier works (Henninger, 1991; Rosson and Carroll, 1996; Michail, 2001; Ye et al., 2000; Ye and Fischer, 2002; Cubranic and Murphy, 2003; Inoue et al., 2003; Hill and Rideout, 2004; Holmes and Murphy, 2005) in software engineering literature focused on developing tools to extract information from software repositories to help developers complete code fragments (e.g., variable, method name or body completion). On the other hand, recent works focus on modeling cross-file information in neural approaches. Wang et al. (2021a) proposed to model intra- and inter-class context for code summarization by extracting the Unified Modeling Language (UML) class diagrams. Shrivastava et al. (2023) proposed a prompt engineering technique that learns a repository-level prompt generator to generate example-specific prompts. Zhang et al. (2023) proposed an iterative retrieval-generation framework to augment prompt with cross-file context. Our work has the same spirit as we propose to retrieve cross-file context given a source code. However, the fundamental difference are 1) we utilize the *import statements* for *structured* retrieval, and 2) we optimize in-file and cross-file context *jointly* in modeling instead of simple prompting.

8. Conclusion

The absence of project-level *cross-file context* for code LMs limits their practicality in modern software development. In this work, we propose CoCoMIC, a framework that incorporates both in-file and cross-file context for code completion based on autoregressive code LMs. We build CCFINDER, a static code analysis tool that builds the project context graph and finds the most relevant cross-file context based on `import` statements. Empirical results show that CCFINDER retrieves 27.07% more context that is not in the current file, and with the retrieved context, CoCoMIC achieves 33.94% relative improvement over the baseline. We further perform various ablations and analysis of various components in CoCoMIC, presenting valuable insights for future research in this direction. Our data and code will be made available upon acceptance.

Ethics Statement

Our work aims at improving code LMs in code generation with cross-file context. We highlight the limitations of our work in the following section. We do not expect our work to have a negative broader impact, though using code LMs always comes with certain risks, e.g., generating biased, toxic, and insecure code. We refer readers to Sec. 7 in (Chen et al., 2021) for a detailed discussion on the broader impact of code LMs.

Limitations

Extension to other languages and third-party packages Our work focuses on Python language, which is widely used and has great availability of open-sourced software projects through PyPI. However, the main concept introduced in our work should be extensible to other languages. In addition, we focus on the project (repo) context in this work, and a potential extension is to incorporate third-party packages and building models to suggest the right third-party libraries to use. We leave these as future work.

Model performances with the absence of cross–file context In this work, we assumed that CoCoMIC could access the other source code files within the project to understand source code dependencies and utilize them accordingly to generate the target code completion. However, CoCoMIC may not access the code files in many cases, *e.g.,* users do not want an AI code LM to read their private or sensitive project APIs. Therefore, it is valid to ask – how CoCoMIC performs when the cross-file context is absent. We evaluate CoCoMIC without access to cross-file context and compare with the finetuned CodeGen model (second row in Table 2). The results show that CoCoMIC performs 5–7% lower (relative performance drop) than the finetuned CodeGen model. Development of training strategies to bridge this performance gap is needed, and we leave this as future work.

Impact on different sized language models Although we use `CodeGen-350-mono` model in this work which consists of 350M parameters, we hypothesize that larger LMs (*e.g.,* 2B, 6B, or 16B variants of CodeGen) would result in similar or higher performance boost due to modeling cross-file context. However, we acknowledge that our work does not substantiate that our proposed technique would boost the performance of LMs of any size.

9. Bibliographical References

Ibrahim Abdelaziz, Julian Dolby, James P McCusker, and Kavitha Srinivas. 2021. A toolkit for generating code knowledge graphs. *The Eleventh International Conference on Knowledge Capture (K-CAP)*.

Ibrahim Abdelaziz, Julian Dolby, Jamie McCusker, and Kavitha Srinivas. 2022. Can machines read coding manuals yet? – a benchmark for building better language models for code understanding. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI 2022)*.

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online. Association for Computational Linguistics.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *ArXiv preprint*, abs/2108.07732.

Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proc. ACM Program. Lang.*, 7(OOPSLA1).

Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. 2021a. Gpt-neo: Large scale autoregressive language modeling with mesh-tensorflow. *If you use this software, please cite it using these metadata*, 58.

Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. 2021b. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow.

Sidney Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, Usvsn Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. 2022. GPT-NeoX-20B: An open-source autoregressive language model. In *Proceedings of BigScience Episode #5 – Workshop on Challenges & Perspectives in Creating Large Language Models*, pages 95–136, virtual+Dublin. Association for Computational Linguistics.

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 *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *ArXiv preprint*, abs/2107.03374.

Davor Cubranic and Gail C Murphy. 2003. Hipikat: Recommending pertinent software development artifacts. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 408–418. IEEE.

Hantian Ding, Jinrui Yang, Yuqian Deng, Hongming Zhang, and Dan Roth. 2022a. Towards open-domain topic classification. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: System Demonstrations*, pages 90–98, Hybrid: Seattle, Washington + Online. Association for Computational Linguistics.

Yangruibo Ding, Luca Buratti, Saurabh Pujar, Alessandro Morari, Baishakhi Ray, and Saikat Chakraborty. 2022b. Towards learning (dis)-similarity of source code from program contrasts. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6300–6312, Dublin, Ireland. Association for Computational Linguistics.

Yangruibo Ding, Luca Buratti, Saurabh Pujar, Alessandro Morari, Baishakhi Ray, and Saikat Chakraborty. 2022c. Towards learning (dis)-similarity of source code from program contrasts. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6300–6312, Dublin, Ireland. Association for Computational Linguistics.

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.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations*.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225, Dublin, Ireland. Association for Computational Linguistics.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Jian Yin, Daxin Jiang, et al. 2021. Graphcodebert: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. 2020. Improved automatic summarization of subroutines via attention to file context. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 300–310.

Scott Henninger. 1991. Retrieving software objects in an example-based programming environment. In *Proceedings of the 14th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 251–260.

Rosco Hill and Joe Rideout. 2004. Automatic method completion. In *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, pages 228–235. IEEE.

Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 837–847. IEEE Press.

Reid Holmes and Gail C Murphy. 2005. Using structural context to recommend source code examples. In *Proceedings of the 27th international conference on Software engineering*, pages 117–125.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Importlab. 2018. importlab: A library for python that automatically infers dependencies and calculates a dependency graph. `https://github.com/google/importlab`.

Katsuro Inoue, Reishi Yokomori, Hikaru Fujiwara, Tetsuo Yamamoto, Makoto Matsushita, and Shinji Kusumoto. 2003. Component rank: Relative significance rank for software component search. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 14–24. IEEE.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652, Brussels, Belgium. Association for Computational Linguistics.

Paul Jaccard. 1912. The distribution of the flora in the alpine zone. 1. *New phytologist*, 11(2):37–50.

Ganesh Jawahar, Benoît Sagot, and Djamé Seddah. 2019. What does BERT learn about the structure of language? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3651–3657, Florence, Italy. Association for Computational Linguistics.

Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Yejin Bang, Andrea Madotto, and Pascale Fung. 2022. Survey of hallucination in natural language generation. *ACM Computing Surveys*.

Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. 2020. Generalization through memorization: Nearest neighbor language models. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Donald E. Knuth. 1984. Literate programming. *Comput. J.*, 27(2):97–111.

Alexandre Lacoste, Alexandra Luccioni, Victor Schmidt, and Thomas Dandres. 2019. Quantifying the carbon emissions of machine learning. *arXiv preprint arXiv:1910.09700*.

Agathe Lherondelle, Yash Satsangi, Fran Silavong, Shaltiel Eloul, and Sean Moran. 2022. Topical: Learning repository embeddings from source code using attention. *ArXiv preprint*, abs/2208.09495.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *ArXiv preprint*, abs/2203.07814.

Yuxian Meng, Shi Zong, Xiaoya Li, Xiaofei Sun, Tianwei Zhang, Fei Wu, and Jiwei Li. 2022. GNN-LM: Language modeling based on global contexts via GNN. In *International Conference on Learning Representations*.

Amir Michail. 2001. Codeweb: Data mining library reuse patterns. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 827–828. IEEE.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA. Association for Computational Linguistics.

D. L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058.

D.L. Parnas, P.C. Clements, and D.M. Weiss. 1985. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11(3):259–266.

Pydeps. 2021. Python module dependency graphs. https://github.com/thebjorn/pydeps.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI preprint*.

Stephen Robertson and Hugo Zaragoza. 2009. The probabilistic relevance framework: Bm25 and beyond. *Found. Trends Inf. Retr.*, 3(4):333–389.

Mary Beth Rosson and John M Carroll. 1996. The reuse of uses in smalltalk programming. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 3(3):219–253.

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 31693–31715. PMLR.

Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. 2001. The structure and value of modularity in software design. ESEC/FSE-9, page 99–108, New York, NY, USA. Association for Computing Machinery.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29.

Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph attention networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax.

Yanlin Wang, Ensheng Shi, Lun Du, Xiaodi Yang, Yuxuan Hu, Shi Han, Hongyu Zhang, and Dongmei Zhang. 2021a. Cocosum: Contextual code summarization with multi-relational graph neural network. *ArXiv preprint*, abs/2107.01933.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021b. 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.

Wikipedia. 2022. Modular programming. https://en.wikipedia.org/wiki/Modular_programming.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10.

Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604.

Yunwen Ye and Gerhard Fischer. 2002. Supporting reuse by delivering task-relevant and personalized information. In *Proceedings of the 24th international conference on Software engineering*, pages 513–523.

Yunwen Ye, Gerhard Fischer, and Brent Reeves. 2000. Integrating active information delivery and reuse repository systems. *ACM SIGSOFT Software Engineering Notes*, 25(6):60–68.

Fengji Zhang, Bei Chen, Yue Zhang, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and

Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*.

Shuyan Zhou, Uri Alon, Frank F. Xu, Zhengbao Jiang, and Graham Neubig. 2023. Docprompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations*.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems*, volume 32, pages 10197–10207. Curran Associates, Inc.