DroidCoder: Enhanced Android Code Completion with Context-Enriched Retrieval-Augmented Generation

Xinran Yu
State Key Laboratory for Novel Software Technology,
Nanjing University
Nanjing, China
xinranyu@smail.nju.edu.cn

Chun Li
State Key Laboratory for Novel Software Technology,
Nanjing University
Nanjing, China
chunli@smail.nju.edu.cn

Minxue Pan*
State Key Laboratory for Novel Software Technology,
Nanjing University
Nanjing, China
mxp@nju.edu.cn

Xuandong Li
State Key Laboratory for Novel Software Technology,
Nanjing University
Nanjing, China
lxd@nju.edu.cn

ABSTRACT

Android is the most popular mobile operating system. However, Android development requires extensive coding, especially for unique features such as lifecycle callbacks and UI widgets. Existing code completion methods typically utilize Retrieval-Augmented Generation (RAG) to provide contextual information for pre-trained code large language models (Code LLMs) to perform completion. Despite considerable progress in these methods, their effectiveness in Android development remains limited. This is because the features of Android development make it challenging for existing retrieval mechanisms to extract sufficient context effectively. In response, we propose DroidCoder, a novel Android code completion framework that employs Android development features and contextual information of code snippets to enrich RAG. It also incorporates a specifically designed loss function to fine-tune the model, enabling it to better utilize context-enhanced RAG for Android code completion. We evaluated our method on three base models and different types of applications, comparing it with two state-of-the-art code completion methods. The experimental results demonstrate that our method significantly outperforms the baselines at line-level and multi-line-level code completion and improves the quality of the completed code.

CCS CONCEPTS

• **Software and its engineering → Software development techniques**.

KEYWORDS

Code completion, Retrieval-augmented generation, Android

*Corresponding author.

ACM Reference Format:
Xinran Yu, Chun Li, Minxue Pan, and Xuandong Li. 2024. DroidCoder: Enhanced Android Code Completion with Context-Enriched Retrieval-Augmented Generation. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695063

1 INTRODUCTION

With the rapid advancement of mobile application development, the Android platform has become one of the most popular mobile operating systems due to its open-source nature, flexibility, and extensive user base [7, 8]. Unlike other software development fields, Android development presents unique challenges. These include managing the lifecycle callbacks of components, interacting with various User Interface (UI) widgets [49], and handling resources and configuration files, all of which require substantial amounts of code.

Recent code completion methods primarily rely on two techniques: large-scale language models (Code LLMs) pre-trained on large-scale code datasets [23, 47, 57] and the Retrieval-Augmented Generation (RAG) approach [30, 55]. RAG enhances language models by finding and leveraging valuable information or examples from external knowledge, particularly in the case of rare patterns. Despite their considerable progress in conventional scenarios, these code completion methods still struggle to address the challenges faced by Android developers. During the pre-training stage, Code LLMs only consider the nearby context of the code to be completed, ignoring the relationships between various program components defined in Android manifest files. Android components such as Activity and Service have specific definitions and dependencies, with their lifecycle callbacks typically being interrelated. However, this information cannot be considered in current Code LLMs used for code completion. Moreover, while current Retrieval-Augmented Generation (RAG) approaches can provide more contextual information, they typically measure the similarity between methods through character-level comparisons. However, this metric faces significant challenges in the Android development environment, with limitations arising mainly from three aspects. First, Android development mostly involves multiple programming languages, each with distinct keywords and syntax structures. This diversity may hinder character-level comparisons in identifying similar functionalities across different programming languages. Second, core function-related methods in Android applications often contain distinctive yet relatively rare tokens that assist in retrieving relevant implementations. For example, functions related to address frequently involve keywords in system calls for GPS. Nevertheless, simple character comparisons struggle to identify these keywords for more relevant retrieval. Third, Android methods extensively use standard UI widgets. The same UI widgets often share similar functional targets, which character-level comparisons may not specifically focus on.

In this paper, we propose DroidCoder, a novel Android code completion framework. DroidCoder utilizes Android development features and contextual information of code snippets to enrich RAG and employs fine-tuning to enable the model to learn how to use the enriched RAG for more effective code completion. The key insight underlying DroidCoder is the Android development features outline the definitions and intents of Android components and UI widgets, which can reflect the functionality of code snippets to some extent. Therefore, utilizing these features can facilitate retrieving functionally similar code. Another key insight is the contextual information of the code to be completed can provide the model with more useful information for completion. Furthermore, by providing both functionally similar code snippets and the context of the code to be completed, the model can learn to utilize the context of the code to be completed by understanding the relationship between similar function code and its context.

To evaluate the effectiveness of DroidCoder, we select four benchmark models with different parameter sizes and architectures and two state-of-the-art RAG-based frameworks, namely RepoCoder [55] and FT2Ra [14]. In both line-level and multi-line-level completion scenarios, DroidCoder significantly improves upon the base models and outperforms the baselines. For example, in the line-level scenario, using CodeGPT as the base model, DroidCoder achieves an improvement of 104.02% in Exact Match (EM) and 28.17% in edit distance (ES). In the multi-line-level scenario, DroidCoder achieves an ES score of 42.46% on the CodeT5+ model, whereas the pure RAG and fine-tuning methods only reached 19.76% and 33.53% on the same model, respectively. Furthermore, following the previous works [19, 46], we also experiment to evaluate the quality of the code generated by DroidCoder. The results indicate that DroidCoder effectively improves the quality of the completed code. Specifically, 93.73% of the code generated by CodeT5+ 770M fine-tuned with DroidCoder has passed our predefined quality checks, significantly surpassing the 29.81% achieved by the original model.

In summary, this paper provides the following key contributions:

- We propose a novel Android code completion framework, DroidCoder, which integrates information retrieved by RAG into the fine-tuning process, enabling the model to better utilize the external code databases for code completion.
- Different from previous RAG, DroidCoder flexibly integrates unique Android development features and contextual information to enhance RAG, thus significantly improving Android code completion performance.
- We demonstrate the effectiveness of DroidCoder through extensive experiments conducted in diverse code completion scenarios with models of different architectures and parameters. The results also indicate that DroidCoder can significantly improve the quality of completed code across different models.

Roadmap. The remainder of this paper is organized as follows: Section 2 introduces Android development features and explains their importance for Android code completion. Section 3 details the design of our proposed framework, DroidCoder. Section 4 describes our experimental setup, and Section 5 presents the results of evaluating DroidCoder in diverse code completion scenarios; Section 6 discusses internal and external threats, while Section 7 reviews recent works on code completion. Finally, Section 8 concludes the paper and discusses future works.

2 BACKGROUND

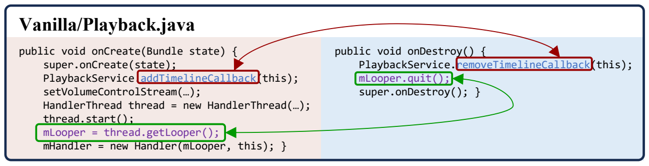
2.1 Android Development Features

We first introduce the features of Android development and discuss why it is essential to incorporate this information into Android code completion.

Configuration Metadata. The Android manifest file includes the core components of each Android application. This file not only declares the permission requirements of each application, such as access to the camera or location data, but also defines Android components like Activities, Services, Broadcast Receivers, and Content Providers. The intent actions that these components can respond to are also indicated in the component definitions through the `intent-filter` tag. This metadata is essential for understanding the relationships between different Android components, thus providing more comprehensive information for Android code completion.

Lifecycle Callbacks. The lifecycle callbacks specify how Activity responds to and switches states based on various system or user events, and they usually share function-related implementations. For example, the *onCreate* and *onDestory* functions correspond to the initialization and release of an Android component. Specifically, as shown in Figure 1, the *TimelineCallback* and *Looper* initialized in the *onCreate* method are released in the *onDestroy* method. Lifecycle callbacks typically have a fixed naming pattern, and the functional targets between the same name methods are similar. Flexible handling of associations and dependencies in the Android lifecycle callbacks is a crucial way to improve Android code completion.

UI Widget Libraries. In addition to the Android standard library, Android development relies on various UI widget libraries [32, 49], such as AndroidX, Material Design, and Jetpack Compose which uses a declarative paradigm to build interfaces through *@Composable* annotations. These libraries provide sufficient interface design elements and interaction patterns. However, employing these libraries requires developers to compose substantial amounts of code related to configuration and event handling, thereby introducing challenges to rapid evolution in Android development [45]. Intuitively, similar widgets often share identical intents. For instance, a left arrow typically relates to back-navigation logic, and



```
public void onCreate(Bundle state) {
    super.onCreate(state);
    PlaybackService.addTimelineCallback(this);
    setVolumeControlStream(…);
    HandlerThread thread = new HandlerThread(…);
    thread.start();
    mLooper = thread.getLooper();
    mHandler = new Handler(mLooper, this); }
```

```
public void onDestroy() {
    PlaybackService.removeTimelineCallback(this);
    mLooper.quit();
    super.onDestroy(); }
```

Figure 1: Example for Associations in Lifecycle Callbacks

the toast is used to display a short message on the screen. Thus, we can auto-complete the UI code by recognizing and referring to the implementations of similar widgets.

Resource Files. Android development also involves static resource information, such as layout files, drawable files, and animation definitions. These resources determine the visual presentation of the application and are embedded in code implementations through identifiers. Specifically, the implementation of the back-navigation method contains references to fade-in and fade-out animations. Similar design goals may use similar resources [58], and these goals are typically related to the functional implementations, hence focusing on these elements can also effectively enhance Android code completion. The resource files and the UI widget libraries constitute the interface metadata.

2.2 Retrieval-Augmented Language Model and Code Completion

According to previous works [51, 55], the code completion enhanced with RAG consists of four steps: indexing, query formation, retrieval, and generation. The main insight underlying this approach is utilizing similar implementations to enhance the pre-trained models to generate more relevant and accurate results. In the indexing stage, a retrieval database is established by partitioning the code files into a collection of code snippets. Section 4.1 provides the specific workflow and details. In the query formation stage, a query is constructed based on the code snippet to be completed. In the retrieval stage, the query constructed before is used for matching similar or relevant snippets in the retrieval databases. The capability and accuracy of the retrieval almost determine the effectiveness of enhancing the outputs of the model. In the generation stage, the snippets retrieved before are concatenated with the code snippet to be completed as a prompt, and the model leverages it to generate more relevant results. However, it is challenging for the Code LLMs to integrate different snippets, and poor results are made due to insufficient contextual information.

3 APPROACH

Figure 2 illustrates an overview of DroidCoder workflow. The framework comprises three stages: Relevant Code Retrieval, Context Enhancement, and Fine-tuning. In the Relevant Code Retrieval stage, we design different retrieval strategies for *component-related* code and other code to search for functionally similar relevant code for code completion in external code databases. Specifically, we design four retrieval rules for component-related code based on Android development features and used tokens that are relatively infrequent but highly functionally distinctive to rank the retrieved methods. The Relevant code retrieval stage can identify potentially functionally similar relevant methods, facilitate the models to better understand the functionality of the code to be completed, and thereby improve completion performance. In the Context Enhancement stage, we provide contextual information for both the functionally similar code and the code to be completed to establish connections between these two code snippets and facilitate the model to better learn how to utilize the knowledge from similar code and the context of code to be completed to complete code. Specifically, we enhance the context from various aspects such as invocation information, lifecycle relationships, resource identifiers, and in-file information. This information can assist the model in better understanding the scenarios in which the relevant code and the code to be completed are used and provide the necessary variables or other contextual information for completion. Finally, in the Fine-tuning stage, we design a novel loss function to compel the model to learn how to utilize the functionally relevant methods and contextual information obtained in the previous two stages for code completion. Through this new loss function, the model can better integrate this knowledge and enhance code completion.

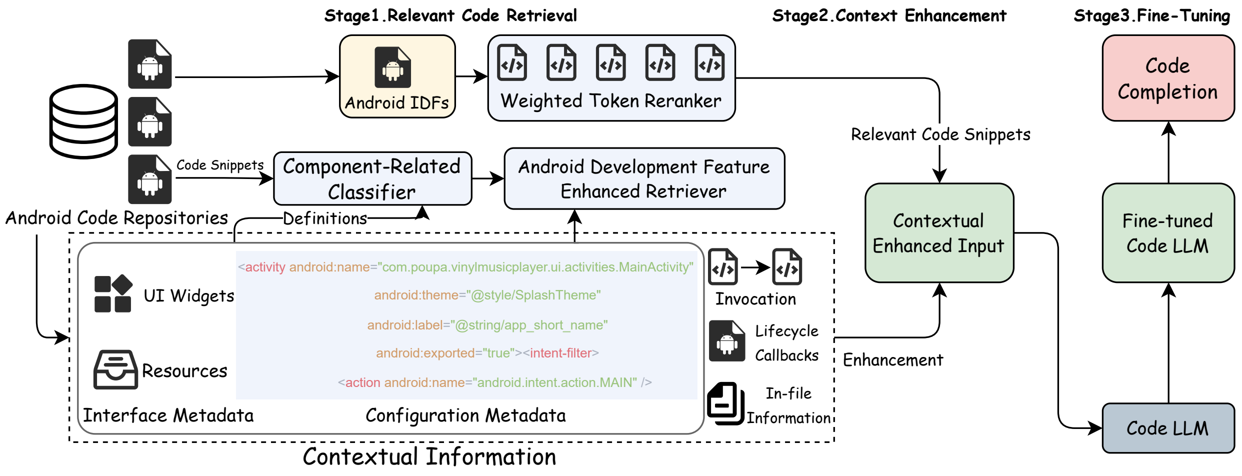
3.1 Relevant Code Retrieval

In the Relevant Code Retrieval stage, DroidCoder utilizes the Android development features to enhance functional relevant code snippet retrieval from Android code databases and ranking for the code snippet to be completed. By retrieving similar methods from external code databases, RAG can enhance the model context and improve code completion. However, the effectiveness of RAG in the context of Android code completion is limited, as character-level similarity comparisons overlook Android features present in the code and eventually lead to sub-optimal retrieval results [42, 55]. Thus, we designed a novel retrieval mechanism that incorporates Android-specific features. Specifically, this stage consists of three parts: component-related classifier, Android development feature enhanced retriever, and weighted token reranker.

Component-Related Classifier. Before retrieving the relevant code, we first construct a classifier to classify whether the code snippets to be completed are *Component Methods* that correspond to Android development features or not, as we have prepared different retrieval strategies for such code to obtain more relevant code comparisons than character-level matching. Specifically, we treat methods involving lifecycle or UI widgets as *Component Methods*, and other methods are treated as *General Methods*. Given a code snippet, the classifier first determines if the method is a lifecycle method of any Android component file defined in the configuration metadata. Then, following the official document, the classifier further checks whether the code snippet relies on the external libraries[1] corresponding to the UI widget. If either of the above conditions is met, the code snippet is considered a *Component Method*; otherwise, it is classified as a *General Method*.

Android Development Feature Enhanced Retriever. After classifying the code snippet to be completed, we select different retrieval strategies based on its type to obtain more relevant code snippets. If the snippet is *Component Method*, we design four rules based on Android development features to retrieve the most relevant code

[1] Android, AndroidX, Material Design, and Jetpack Compose.



```
<activity android:name="com.poupa.vinylmusicplayer.ui.activities.MainActivity"
android:theme="@style/SplashTheme"
android:label="@string/app_short_name"
android:exported="true"><intent-filter>
<action android:name="android.intent.action.MAIN" />
```

Figure 2: Approach Overview of DroidCoder

snippets from the code databases. If the snippet is *General Method*, we use the default character-level comparison for retrieval.

Specifically, given a code snippet to be completed that belongs to *Component Method* and a code snippet from the code databases, we compare their similarity based on their relationship with Android components (Rules 1-2), whether they are the same lifecycle function (Rule 3), and the involved UI widgets (Rule 4). The underlying insights are threefold. First, configuration metadata defines the intents that components can respond to, which reflects the components' functionality. Therefore, components that belong to the same component type or can respond to the same intent are more relevant. Second, the same type of lifecycle callback is always invoked in similar scenarios, making them more similar. Finally, the use of similar UI widgets indicates similar functional requirements and makes them more related. More specifically, **Rule 1** is to examine whether they belong to the same type of Android components (e.g. activity, provider). **Rule 2** is to see whether the actions under the $<intent\text{-}filter>$ tag of the components they belong to are the same. **Rule 3** is to compare if the two snippets are both lifecycle methods and if they are lifecycle methods with the same name. **Rule 4** is to compare whether they have the same base and UI libraries imported, these libraries are Android, AndroidX, Material Design, and Jetpack Compose library as mentioned earlier. The more rules that are satisfied, the more similar we consider the two code snippets to be.

For a code snippet to be completed that belongs to *General Method*, we combine two similarity measurements that are widely used in the existing works [42, 55] to locate similar implementations from different perspectives. The one is BM25 [39], often used for computing lexical similarity between the query and document. And the other one is Jaccard [18], which focuses on the proportion of shared words.

Weighted Token Reranker. After retrieving relevant code snippets for the code snippet to be completed, we design a novel ranking algorithm based on functional distinctive tokens in methods to rerank these relevant code snippets. This is because the retrieval of relevant code primarily considers Android development features and does not account for the functional similarity related to the code logic. Intuitively, different tokens contribute differently when distinguishing the functionality of various code snippets. Specifically, some tokens that are relatively infrequent across the entire code database may have strong functional distinctiveness. For example, location-related functions (motion track recording, location sharing, etc.) are accompanied by frequent occurrences of GPS-related tokens, as they all need to invoke the system permission of GPS. However, those GPS-related tokens rarely appear in network-related code snippets. Thus, we can reorder methods functionally using these relatively infrequent but functional distinctive tokens. Our approach also effectively addresses the issue of different programming languages having distinct keywords and syntax structures in Android development, as these functionally distinctive tokens are common across different programming languages.

Table 1: Selected Top Apps For IDF

App	#Stars	#Commits	App	#Stars	#Commits
AmazeFileManager	5077	6496	Anki-Android	7911	19106
muzei	4644	2936	WordPress-Android	2916	83560
NewPipe	28796	11214	firefox-android	1631	31293
Rocket.Chat	38954	25564	Signal-Android	24949	14643
ExoPlayer	21497	18873	Joplin	42663	11336

Specifically, we utilize the TF-IDF (Term Frequency-Inverse Document Frequency) [5] to realize our idea. TF-IDF is a numerical statistic that reflects the importance of a token in a document within a collection of documents. The importance of a token in a document is directly proportional to its frequency within that document but inversely proportional to its frequency across the entire corpus. In our context, we treat each code snippet as a document and collect the representative open-source APPs as the entire corpus. Then, we employ the TF-IDF value (i.e., importance) of a token as its functional distinctiveness. Specifically, we collect ten APPs with more than 4k stars or 10k commits that are still maintained recently. Table 1 includes their details. Then we extract all the methods of each APP and calculate the IDF value of each token t in equation 1, where N denotes the total number of methods and n_t is the count of methods containing the token t.

$$IDF(t) = log(\frac{N}{n_t}) \tag{1}$$

Given a code snippet to be completed and a relevant code snippet, we first calculate the TF-IDF value of each token as $F_{ti}(t) = freq(t) * IDF(t)$ to present the functional distinctiveness of each token, where $freq(t)$ counts the occurrence of t in code snippets. For the token that is not included in IDF calculation, we use fuzzywuzzy [2] to find the most similar token as a substitute. Fuzzywuzzy compares the similarity of two tokens using the Levenshtein distance algorithm to measure the edit distance between them, and we choose it for its effectiveness in identifying alternatives that share the same root word and its impressive efficiency.

After obtaining the functional distinctiveness of each token, we next compare the functional similarity between the code snippet to be completed and the relevant code snippet. Intuitively, if the overlapping tokens between the two methods have higher functional distinctiveness and the non-overlapping tokens have lower values, we consider the two methods to be more similar. Specifically, let S_q and S_c represent the token set of the code snippet to be completed and the relevant code snippet, the non-overlapping token set is calculated in equation 2. Then, we calculate the similarity between these two code snippets as equation 3.

$$S_q \triangle S_c = (S_q \cup S_c) \setminus (S_q \cap S_c) \tag{2}$$

$$score = \frac{\sum_{s_i \in (S_q \triangle S_c)} F_{ti}(s_i) - \sum_{s_j \in (S_q \cap S_c)} F_{ti}(s_j)}{\sum_{s_k \in (S_q \cup S_c)} F_{ti}(s_k)} \tag{3}$$

Through equation 3, the more important the overlapping tokens are and the less important the non-overlapping tokens are, the more similar the methods are. We use normalization to eliminate the impact of the number of tokens. Algorithm 1 presents the process of relevant code retrieval. Given a code snippet $\mathcal{F}$ to be completed, We first classify it (Line 1). Then, if the $\mathcal{F}$ is *General Method*, we utilize BM25 and Jaccard to retrieve the Top-K relevant code snippets (Lines 2-5). Otherwise, if the $\mathcal{F}$ is *Component Method*, we employ four Android development feature-related rules to retrieve the Top-K relevant code snippets (Lines 6-9). Finally, we rerank the relevant code snippets based on functional distinctive tokens (Line 10).

Algorithm 1 Relevant Code Retrieval

```
Input: Snippet to be Completed $\mathcal{F}$, Code Retrieval Databases $\mathcal{D}$
Output: ReRanked Relevant Results $R$
1:  $\mathcal{F}.type \leftarrow classify(\mathcal{F})$
2:  if $\mathcal{F}.type$ == "General Method" then
3:      $sim_{bm} \leftarrow BM25(\mathcal{F}, \mathcal{D})$                ▷ Calculate BM25 similarity
4:      $sim_{jac} \leftarrow Jaccard(\mathcal{F}, \mathcal{D})$           ▷ Calculate Jaccard similarity
5:      $relevant \leftarrow Rank(sim_{bm}, sim_{jac}, TopK)$      ▷ $TopK$ Results
6:  else if $\mathcal{F}.type$ = "Component Method" then
7:      $sim \leftarrow multiAndroidSim(\mathcal{F}, \mathcal{D})$        ▷ Android Rules Similarity
8:      $relevant \leftarrow Rank(sim, TopK)$                     ▷ $TopK$ Results
9:  end if
10: $R \leftarrow reRank(\mathcal{F}, relevant)$                    ▷ Rerank $TopK$ Results
11: return $R$
```

3.2 Context Enhancement

During the Context Enhancement stage, we provide in-project context for both the functionally similar code and the code to be completed, making the model understand the scenarios between them and the necessary information for completion (such as global variables). This is because the functionally similar code provided in the previous stage is extracted from other code applications and may differ significantly from the in-project context of the code to be completed. Therefore, we need to provide the in-project context of these two parts to assist the model in establishing connections between them. Otherwise, without the in-project context of relevant methods and the code to be completed, the model may tend to directly clone instead of learning how to transfer the knowledge of functionally similar code to the code to be completed. For example, paired lifecycle callbacks in Android can offer useful information for code completion, such as *onCreate* methods could provide useful context for *onDestroy* methods. Specifically, we provide in-project context for the code completion from four perspectives: invocation information, lifecycle relationships, resource identifiers, and in-file information.

Invocation information. The caller methods of the functionally similar code (i.e., the callee method) can provide more context information which facilitates the model to understand the scenario of the functionally similar code, thereby enhancing code completion.

Paired lifecycle callbacks. Lifecycle callbacks mostly appear in pairs and are interrelated. For instance, the resources released in the *onDestroy* method are typically initialized in the *onCreate* method. Thus we use the symmetric method for each lifecycle method.

Resource identifiers. Users interact with Android applications through the interface, making drawable, layout, and animation resources highly frequently used. If methods have similar functional intents, they may load similar types of resource identifiers. Utilizing this, we directly query within the project for methods with the same name and provide their used resource identifiers to the code to be completed.

In-file information. It is worth noting that in-file details, including other methods, global variables, and libraries, can provide direct information that is involved in the results. Specifically, we first calculate the method signatures and global variables within the file that have the highest semantic similarity to the name of the code to be completed and provide them as context to the model. We employ the cutting-edge calculation method Sentence-BERT [38], which is widely used in previous works [13, 33, 54] to match semantic similarity, as the method and variable names are consistent with human reading habits. Finally, we also provide the external libraries used within the file to the model.

After obtaining all the context for the code to be completed, the next step is fine-tuning the model using this information. Following previous works [55], we organize this information into training data using templates. Specifically, we structure the training data in the order of relevant code, context information, task instructions that indicate code completion, and code to be completed.

3.3 Fine-tuning

In the fine-tuning stage, we utilize the previously prepared training data to fine-tune the model. Specifically, the training data consists of two parts: the code snippet to be completed with the task instructions (denoted as C), and the relevant code snippets and context information (denoted as M). The underlying insight of including M in the fine-tuning process is that learning how to utilize the information in M can improve code completion intuitively. To realize our idea, we design a novel loss function to encourage the model to utilize the information in M.

Specifically, we compare the loss value before and after adding M. If the inclusion of M results in an increased loss, it indicates that the model is not effectively utilizing the information in M. In this case, We increase the coefficient of the loss function (i.e., increase the loss) to make the model pay more attention to how to integrate the information from M. More specifically, let $\mathcal{L}(C)$ represent the loss on C and $\mathcal{L}(M \cup C)$ represent the loss on M and C, where $\mathcal{L}$ is the cross-entropy loss. The final loss is given below as follows:

$$\mathcal{L} = (1 + max(\mathcal{L}(M \cup C) - \mathcal{L}(C), 0)) \times \mathcal{L}(M \cup C) \tag{4}$$

In equation 4, we quantify the difference between the two loss functions to adjust the coefficient of $\mathcal{L}(M \cup C)$. $\mathcal{L}(M \cup C) > \mathcal{L}(C)$ indicates that the model is not effectively utilizing the information in M. Thus, we increase the coefficient to make the model focus more on learning to integrate M. On the contrary, when $\mathcal{L}(M \cup C)$ is not greater than $\mathcal{L}(C)$, the result of the loss function is equal to $\mathcal{L}(M \cup C)$.

4 EXPERIMENTAL SETUP

We evaluate DroidCoder on the following research questions:

- **RQ1:** How effective is DroidCoder in different Android code completion scenarios?
- **RQ2:** How do different components within DroidCoder affect its overall effectiveness?
- **RQ3:** Can DroidCoder generate higher-quality code compared to existing methods?

4.1 Benchmarks

Datasets. Android apps iterate and update quite rapidly, making it challenging to keep the previous datasets up to date with Android library APIs. Also, some open-source apps have converted to closed-source or stopped maintenance in previous datasets. Therefore, we carefully construct our benchmark by following the workflow in Repocoder [55]. We randomly select 78 Android app repositories from Github that satisfy the following criteria: more than 200 stars, open-source license, primarily written in Java or Kotlin, and actively maintained [53], specifically with persistent commits in recent six months. Complete copies of the repositories were archived for analysis and experiments as of February 2024. Additionally, our dataset includes applications of various scales, including large-scale applications with around 20K lines of code.

Following the previous work [42], retrieving relevant information is more efficient within similar applications. Therefore, we perform retrieval only from the code repositories of similar types of applications, which can greatly improve the efficiency of retrieval. We classify these applications based on descriptions from their official repositories and their tag in Google Play. To ensure sufficient training data [36, 37] and avoid potential class imbalance issues during fine-tuning [44, 56], we selected categories with at least 10 applications and prepared 10 apps with the highest number of stars and commits for each category for the experiments. Specifically, we selected the following four types: communication, life, notes, and multi-media.

Table 2: Experimental Subjects

ID	APP	CATEGORY	TAG	Main Language
A1	Infinity-For-Reddit	Communication	train	Java
A2	thunderbird-android	Communication	train	Kotlin
A3	FairEmail	Communication	train	Java
A4	IRCCloud	Communication	train	Java
A5	tutanota	Communication	train	Kotlin
A6	Simple-Contacts	Communication	train	Kotlin
A7	android-oss	Communication	train	Kotlin
A8	Tusky	Communication	test	Kotlin
A9	RedReader	Communication	test	Java
A10	Xtra	Communication	test	Kotlin
A11	neutrinote	Notes	train	Java
A12	Notally	Notes	train	Kotlin
A13	Omni-Notes	Notes	train	Java
A14	Simple-Notes	Notes	train	Kotlin
A15	MyBrain	Notes	train	Kotlin
A16	tasks	Notes	train	Kotlin
A17	notes-android	Notes	train	Java
A18	aaf-easydiary	Notes	test	Kotlin
A19	simplenote-android	Notes	test	Java
A20	Compose-ToDo	Notes	test	Kotlin
A21	Noice	Multi-Media	train	Kotlin
A22	AntennaPod	Multi-Media	train	Java
A23	jellyfin-android	Multi-Media	train	Kotlin
A24	RetroMusicPlayer	Multi-Media	train	Kotlin
A25	vlc-android	Multi-Media	train	Kotlin
A26	SeriesGuide	Multi-Media	train	Kotlin
A27	Voice	Multi-Media	train	Kotlin
A28	Simple-Gallery	Multi-Media	test	Kotlin
A29	cloudstream	Multi-Media	test	Kotlin
A30	vanilla	Multi-Media	test	Java
A31	money-manager-ex	Life	train	Java
A32	Catima	Life	train	Java
A33	unstoppable-wallet	Life	train	Kotlin
A34	runnerup	Life	train	Java
A35	uhabits	Life	train	Kotlin
A36	Transportr	Life	train	Kotlin
A37	Simple-Calendar	Life	train	Kotlin
A38	habitica-android	Life	test	Kotlin
A39	OpenTracks	Life	test	Java
A40	Photok	Life	test	Kotlin

Before fine-tuning, we randomly divide each type of application into a 7:3 ratio for training and testing sets. During relevant code retrieval, we perform the retrieval only from the codebase constructed from the training set. This aligns with the practical scenario in the early development stage where the current application under development lacks sufficient methods and requires a substantial amount of code. Also, considering that most models are not pre-trained under the Kotlin language projects, we ensure at least two of the test apps in each category are written in Kotlin to avoid performance bias. All details of the subject app repositories can be found in Table 2.

Completion Scenarios. Following previous code completion benchmarks and android features [14, 42, 55], we consider line-level code completion as well as the more challenging and realistic scenario of multi-line-level code completion.

Specifically, we first extract all methods from each app to cover different cases as much as possible with the help of JavaParser [15], KasTree [1], and Kotlinx [3]. Following previous studies [8, 43], we discard methods with bodies less than 3 lines or more than 60 lines, which account for about 5% of all methods. Analyzing all the remaining methods in the training set, the median length of the method body is 10, and the average is 11.08. Therefore, considering the programming habits of developers and eliminating the random bias, we always set the last 5 lines of the method body (or all of them if the method body is less than 5 lines) as the multi-lines to be completed. To avoid the complexity of random selection and ensure consistency, we evaluate line-level completion by considering only the first line of the multi-line results. Additionally, for line-level completion, we followed previous work [55] that used only single lines of code that contain at least five tokens and are not comments.

Finally, we double-check each test case to make sure there are no exact duplicate samples, avoiding performance bias, and removing all cases with @*Deprecated* annotation. After filtering, the training dataset comprising approximately 1.5 million lines of code, 10k files, and 30k methods. And the test dataset contains about 428k lines of code, 3k5 files, and 5k methods.

Cross-category Scenario. To conduct a more comprehensive evaluation, we considered cross-category scenarios. This involves using models trained on one type of application to perform code completion on a test set from a different type of application. This more challenging scenario typically occurs when the external databases contains only a small portion of code snippets closely relevant to the application being developed. And this evaluation includes about 500k lines of code, 5k files, and 5k methods.

4.2 Baselines

Base Models. To better examine our approach, we use code LLMs[57] with diverse architectures[46] and parameter scales, which are widely used in previous works to fine-tune and evaluate.

- **CodeT5+ [47]**: CodeT5+ belongs to the encoder-decoder model category. We select two versions that comprise 220 million or 770 million model parameters.
- **CodeGPT [31]**: CodeGPT is a GPT-based model, specifically designed for code-related tasks. It is worth noting that CodeGPT-adapted, has been widely used in previous works [14, 42], thus we select it for more comparative experiments.
- **StarCoder [23]**: StarCoder is also decoder-only code LLM and pre-trained on 1 trillion tokens sourced from the Stack. Considering our Android scenario, we conduct experiments on StarCoderBase-1B, a model trained on over 80 programming languages.

Compared Approaches. Besides, we also compare DroidCoder with two state-of-the-art code completion approaches:

- **RepoCoder [55]**: RepoCoder is a state-of-the-art RAG approach and makes effective use of repository-level information for code completion. To ensure consistency, we choose the same model, GPT-3.5-Turbo, as in RepoCoder. It is a commercial generation model with billions of trainable parameters and has been pre-trained on an extensive code corpus.
- **FT2Ra [14]**: FT2Ra is the latest retrieval-augmented code completion approach inspired by fine-tuning and outperforms previous baselines. Similarly, we compare our results with CodeGPT-adapted, as this model was selected by the original paper.

Configurations. To ensure fairness, we perform full-parameter fine-tuning for CodeGPT and CodeT5+. For StarCoder, we use parameter-efficient fine-tuning with LoRA [16] due to resource limitations, while the r is set to 8 and α is set to 16 to follow the original paper [23]. For the outputs except ChatGPT, we use beam search instead of sampling to ensure the results are idempotent and reproducible, with the number of beams set to 5, as suggested by previous works. According to the empirical results (c.f. Section 5.2), we set $TopK$ = 10. All experiments are conducted on a workstation with Intel(R) Xeon(R) Gold 6133, 128GB memory, and two RTX 4090 GPUs, running Ubuntu 20.04.

Evaluate Metrics. Following previous works [14, 30, 42, 55], we choose *Exact Match* accuracy (EM) and *Levenshtein Edit Similarity* (ES) [21] as metrics for code completion. In multi-line code completion, the EM metric requires that every line is perfectly matched. In practice, we return the percentage of successful line-level exact matches as EM score in multi-line code completion. We denote it as EM* in the multi-line level scenario.

5 EXPERIMENTAL RESULTS

5.1 RQ1: Effectiveness on Completion Scenarios

In this RQ, the primary objective is to evaluate the effectiveness of DroidCoder in code completion on different completion scenarios. We conduct a comparative analysis of DroidCoder against three base models and two state-of-the-art code completion approaches on 40 APPs outlined in Section 4.1 and Section 4.2, examining performance at both line and multi-line level. Results are detailed in Table 3 and Table 4. Each row in the table represents the completion performance of different methods or models on the same application. Each column represents the completion performance of the same method or model across all applications.

Line-level Completion. From the Table 3, we have the following observations. First, compared to the base models, our method significantly improves the code completion performance on Android code. Specifically, compared to the original CodeT5+700M, StarCoderBase-1B, and CodeGPT models, after fine-tuning by our method, the average EM and average ES of them across all test apps increased from 5.66 to 19.85 and 18.73 to 42.69 respectively. Second, our method outperforms the state-of-the-art code completion methods. Notably, compared to FT2Ra, CodeGPT fine-tuned with our method achieves an average improvement of 104.02% in EM and 28.17% in ES. This result demonstrates the effectiveness of DroidCoder at line-level Android code completion. This also suggests that DroidCoder has a better retrieval capability. Unlike FT2Ra, which relies solely on character-level similarity for retrieval, DroidCoder enhances the retrieval process using Android development features to extract more functionally relevant code snippets. By further ranking these snippets with functionally distinctive tokens, DroidCoder provides the model with more functionally similar code snippets in the context of Android code completion.

Multi-line-level Completion. Table 4 presents the results for multi-line-level completion. From the table, we can observe a similar trend to line-level completion. Specifically, DroidCoder improves the performance on three models across all test apps. After fine-tuning CodeT5+770M using DroidCoder, the average EM* and average ES across different applications improved by 237.32% and 185.08% respectively. Additionally, we observed that the original CodeGPT model achieved only 0.44 in EM* and 7.59 in ES, which is a decrease of 98.36% and 61.59% compared to RepoCoder-enhanced ChatGPT. However, after fine-tuning with our method, CodeGPT showed an improvement of 27.50% in EM* and 35.98% in ES compared to RepoCoder+ChatGPT. This further demonstrates that our method can effectively enhance the model's performance in multi-line code completion. The relevant code retrieval and context enhancement can also improve the performance of code completion in more challenging scenarios that multi-line code to be completed, accelerating the development of new apps from scratch.

Cross-category Scenario Completion. we further evaluate the multi-line-level effectiveness of DroidCoder in a cross-category

Table 3: Results of Line-level Completion(%)

Repo.	DroidCoder								FT2Ra		RepoCoder		Original					
	CodeT5+ 770M		CodeT5+ 220M		StarCoderBase-1B		CodeGPT		CodeGPT		ChatGPT		CodeT5+ 770M		StarCoderBase-1B		CodeGPT	
	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES
aaf-easydiary	**24.78**	**50.08**	22.17	46.87	20.87	49.55	12.61	37.96	2.44	28.80	2.17	19.73	8.70	24.84	10.43	30.93	0.43	5.31
Compose-ToDo	14.12	**46.57**	11.18	45.34	**14.71**	45.21	7.06	26.82	2.33	8.45	2.35	20.78	5.88	26.54	0.59	15.73	0.58	6.39
simplenote-android	**25.27**	**51.26**	24.46	48.10	21.47	42.94	17.12	38.80	11.88	33.72	4.08	20.67	18.75	35.26	0.82	16.20	0.27	6.58
cloudstream	**18.80**	**42.98**	18.29	41.72	18.12	40.78	7.86	25.47	0.93	17.56	1.20	16.07	7.86	23.57	2.74	19.04	0.17	4.95
Simple-Gallery	**26.30**	**49.68**	21.43	43.18	21.10	40.77	10.39	27.90	1.64	18.27	1.95	16.77	4.55	25.24	6.17	21.36	0.32	5.52
vanilla	**30.71**	**54.18**	26.63	51.66	23.10	48.06	17.39	38.42	11.63	32.07	6.79	21.88	13.59	29.87	0.27	11.46	0.27	5.56
habatica-android	**25.91**	**47.42**	23.75	46.06	23.08	44.54	12.01	27.92	0.81	18.48	2.43	18.23	9.58	26.88	6.21	22.58	0.13	6.10
OpenTracks	**25.42**	**52.59**	23.75	49.54	22.33	48.46	16.86	41.04	12.82	40.00	5.94	22.59	14.96	30.14	4.75	19.07	0.24	6.00
Photok	**27.27**	51.60	24.68	50.20	27.27	**52.90**	9.09	29.00	2.56	14.19	3.90	22.57	4.55	23.58	9.09	27.04	0.65	5.87
RedReader	28.79	**52.59**	27.97	51.07	**31.51**	52.00	22.51	42.10	10.10	37.80	4.77	20.34	22.78	38.99	0.41	15.09	0.14	5.77
Tusky	**22.69**	**48.86**	20.38	44.70	20.38	43.90	5.77	21.35	5.62	22.84	5.96	23.46	5.96	21.24	16.35	41.75	0.19	5.55
Xtra	**25.29**	**45.77**	22.38	43.20	22.97	42.57	13.66	34.68	11.86	33.27	4.36	20.95	5.23	23.85	19.77	34.26	0.29	6.19
Avg.	**24.61**	**49.47**	22.26	46.80	22.24	45.97	12.69	32.62	6.22	25.45	3.83	20.33	10.20	27.50	6.47	22.88	0.31	5.82

Table 4: Results of Multi-line-level Completion(%)

Repo.	DroidCoder								FT2Ra		RepoCoder		Original					
	CodeT5+ 770M		CodeT5+ 220M		StarCoderBase-1B		CodeGPT		CodeGPT		ChatGPT		CodeT5+ 770M		StarCoderBase-1B		CodeGPT	
	EM*	ES	EM*	ES	EM*	ES	EM*	ES	EM*	ES	EM*	ES	EM*	ES	EM*	ES	EM*	ES
aaf-easydiary	**52.15**	**45.14**	49.32	40.59	44.16	42.22	33.36	27.65	9.74	18.44	24.12	19.30	16.94	17.14	12.39	11.66	0.46	7.84
Compose-ToDo	**54.16**	**46.48**	51.61	42.51	44.90	36.86	34.42	24.70	11.40	12.53	23.46	17.77	18.29	16.58	11.19	9.28	0.30	6.46
simplenote-android	**55.24**	**47.98**	51.63	42.08	43.35	36.86	30.11	23.10	15.22	20.02	28.76	22.54	19.57	16.61	13.48	10.59	0.68	7.99
cloudstream	**53.12**	**41.64**	50.60	38.98	44.98	37.47	31.52	22.54	10.46	14.68	22.73	15.68	16.80	15.24	11.56	9.44	0.12	6.51
Simple-Gallery	**57.06**	**46.19**	51.40	39.21	45.06	34.04	31.97	21.27	11.37	17.95	25.65	16.55	17.35	15.21	12.44	8.90	0.90	6.94
vanilla	**53.43**	**47.56**	48.85	45.74	43.56	40.99	36.67	33.03	13.95	18.54	24.40	21.25	16.11	17.14	11.41	10.40	0.39	8.27
habatica-android	**55.73**	**45.30**	53.17	42.17	46.03	38.01	30.11	22.59	12.70	16.30	28.75	23.03	17.04	17.24	12.49	10.17	0.40	7.60
OpenTracks	**52.68**	**46.89**	49.67	44.90	45.26	41.47	38.18	32.85	17.72	21.98	27.36	23.03	13.77	15.73	10.73	11.22	0.40	8.87
Photok	**55.30**	**47.66**	49.92	42.63	48.68	45.44	27.88	22.81	10.51	17.46	24.86	20.10	15.00	16.35	14.44	11.30	0.41	6.79
RedReader	**57.26**	**49.41**	53.96	46.12	50.57	45.33	43.00	33.68	16.21	20.73	28.75	21.41	18.35	16.60	14.79	10.98	0.70	8.13
Tusky	**54.66**	**46.20**	50.47	41.01	45.61	38.73	23.69	18.84	10.79	15.52	26.90	21.69	12.22	14.10	20.23	16.30	0.21	7.55
Xtra	**56.55**	**44.83**	54.34	41.72	49.16	39.45	39.94	29.35	11.23	33.61	27.73	20.23	15.49	16.76	12.48	9.84	0.26	8.13
Avg.	**54.78**	**46.27**	51.22	42.46	46.44	40.05	34.26	26.87	12.61	18.98	26.87	19.76	16.24	16.23	13.03	10.82	0.44	7.59

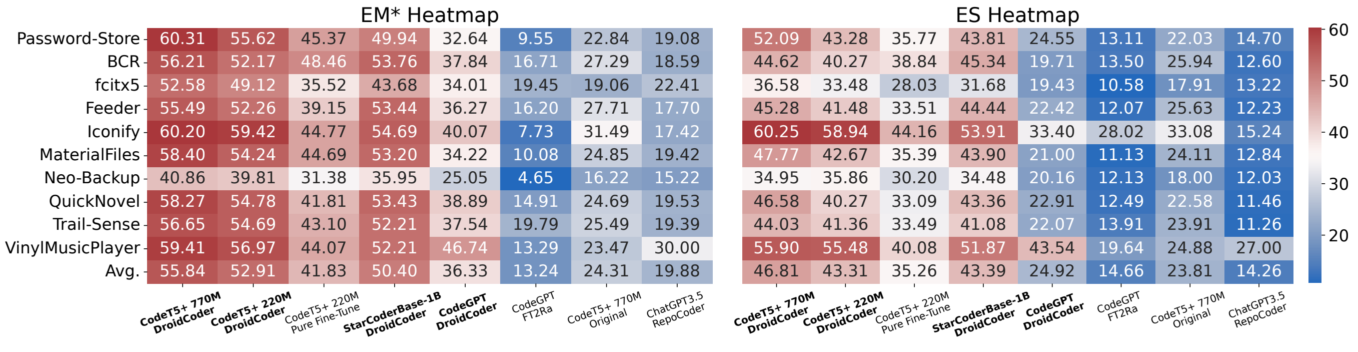


Figure 3: Results on multi-line-level code completion under cross-category scenarios(%).

scenario. This is a more challenging scenario and one that is likely to occur in practice because the external code databases may contain only a small portion of code snippets relevant to the application being developed. Specifically, from the 78 collected applications, we selected 10 with the highest star and commit counts from each of the other types. Figure 3 presents the results of different baselines on different apps. The color gradient from blue to white to red indicates performance improvement, with deeper red representing better performance. Specifically, the CodeT5+770M model fine-tuned with DroidCoder showed an increase in the average EM* score from 19.88% to 55.84% compared to the original model. Compared to the state-of-the-art fine-tuning method FT2Ra, our method improved the average ES score by 69.98% when fine-tuning CodeGPT.

Case Study on multi programming languages. Tables 3 and 4 indicate that our method can perform code completion for applications written in Java or Kotlin. Specifically, we also found that our method enables the model to complete code by utilizing functionally similar functions from another code database. Figure 4 presents a case in which the code to be completed is written by Kotlin but relevant code is written by Java. From the figure, we found that our method (using CodeT5+ 770M as the base model) successfully utilizes Java code to complete functions in Kotlin. However, commercial models like Claude3-Haiku [4] and the original base model CodeT5+ 770M both fail in this task.

ID: Xtra/491

```
// code snippet to be completed
override fun observe(owner: LifecycleOwner, observer: Observer<in T>) {
        if (hasActiveObservers()) {
            Log.w(TAG, "Multiple observers registered but only one will be
notified of changes.")
        }
        super.observe(owner, Observer<T> { t ->
```

```
// Reference Snippet:
@MainThread
public void observe(@NonNull LifecycleOwner owner, @NonNull final
Observer<? super T> observer) {
    if (hasActiveObservers()) {
        Timber.w("Multiple observers registered but only one will be
notified of changes.");
    }
    // Observe the internal MutableLiveData
    super.observe(owner, new Observer<T>() {
        @Override
        public void onChanged(@Nullable T t) {
            if (pending.compareAndSet(true, false)) {
                observer.onChanged(t);
            }
        }
    });
}
```

```
// Ground Truth and DroidCoder(CodeT5+ 770M) output:
            if (pending.compareAndSet(true, false)) {
                observer.onChanged(t)
            }
        })
    }
```

```
// Claude3-Haiku:
    if(!owner.lifecycle.currentState.isAtLeast(Lifecycle.State.STARTED)) {
                return@Observer
            }
            observer.onChanged(t)
        })
    }
// CodeT5+ 770M:
            observer.onNext(t)
        } })
        return this
}
```

Figure 4: Example for different programming languages.

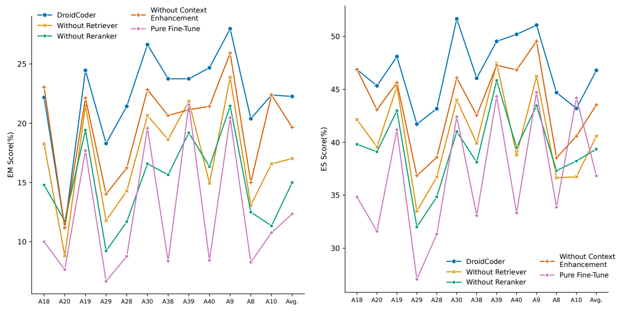


Figure 5: Ablation study at line-level completion.

Answer to RQ1: DroidCoder outperforms all the baselines and base models for line-level and multi-line-level code completions. In cross-category scenarios, DroidCoder also achieves the best performance.

5.2 RQ2: Ablation Study

Contribution of Each Component. We first study the contribution of different components in DroidCoder Specifically, we consider four variants of DroidCoder: without Retriever, without Reranker, without Context Enhancement, and without RAG during Fine-tuning (i.e., pure fine-tuning). To ensure consistency while

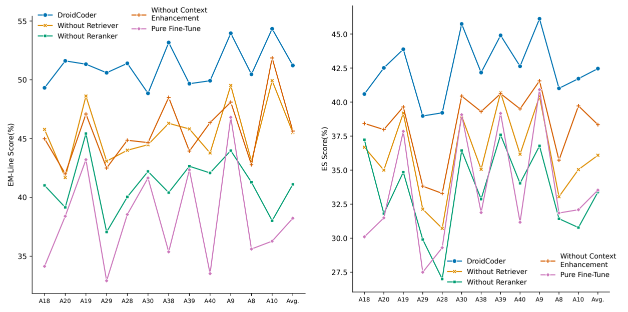


Figure 6: Ablation study at multi-line-level completion.

taking into account computational resources, we select CodeT5+ 220M for ablation studies. Figure 5 and Figure 6 present the ablation experimental results for each component of DroidCoder. The different lines in the chart represent various variants of the DroidCoder. The x-axis represents different apps in the test set, with the final point on the x-axis showing the average of all results. From the figures, we have the following observations.

First, each module contributes to the effectiveness of DroidCoder. When any single component is removed, the effectiveness of DroidCoder decreases. Specifically, removing the retriever, reranker, and context enhancement, and using only fine-tuning leads to an average decrease in EM score at line-level code completion by 23.54%, 32.17%, 13.22%, and 44.52%, respectively. **Second**, reranking the retrieval results using relatively infrequent but highly distinctive tokens is crucial. Specifically, from the figures, we can observe that removing the reranker results in DroidCoder achieving the worst or second-worst performance on average across different scenarios. The results may suggest that the functionality of Android methods exhibits token cohesion. By using specific tokens, we can effectively cluster different methods, allowing us to retrieve methods that are more functionally similar. **Third**, we found that removing the retriever always results in a larger performance decline than removing context enhancement across different scenarios. This demonstrates that functionally similar functions contribute more significantly to code completion and the model tends to use or imitate functionally similar functions during completion rather than relying on the provided context. This phenomenon may suggest relevant code snippets are easier for the code model to understand, whereas the provided context may be more loosely structured (including variables, identifiers, and code snippets), making it relatively difficult for the model to utilize this information effectively. How to better utilize this part of the information will be the focus of our future work. **Finally**, incorporating RAG during fine-tuning is essential, as using fine-tuning alone almost leads to the worst results in the figures. This may be explained that integrating enhanced knowledge during fine-tuning significantly improves the model's ability to utilize this knowledge during inference.

Impact of the Choice of TopK. To evaluate the performance of DroidCoder by the value of *TopK*, we set it to 5, 10, 20, and 50 as the previous work [14] suggested. The results are presented in Figure 7, illustrating that the choice of *TopK* is not highly sensitive to the completion results. The smaller *TopK* may lead to the neglect of the function-related tokens, while a larger *TopK* makes the approach heavily dependent on the quality of the IDF dictionary.

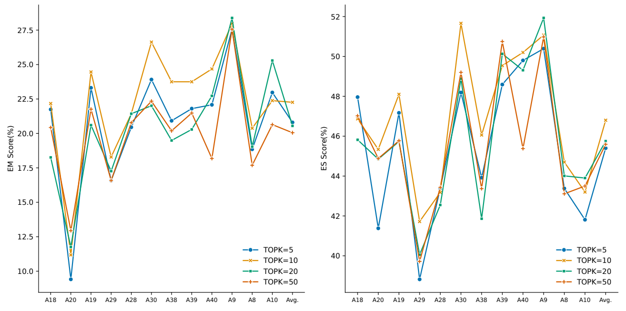


Figure 7: Results at line-level completion under different Topk.

Table 5: Compilation Assurance Rate

Model	Positive Rate
CodeT5+ 770M + **DroidCoder**	**93.73%**
CodeT5+ 220M + **DroidCoder**	91.50%
StarCoderBase 1B + **DroidCoder**	84.99%
ChatGPT3.5 Pure RAG (RepoCoder)	83.53%
CodeT5+ 220M Pure Fine-Tune	65.78%
CodeT5+ 770M	29.81%
StarCoderBase 1B	18.86%
CodeGPT	5.71%

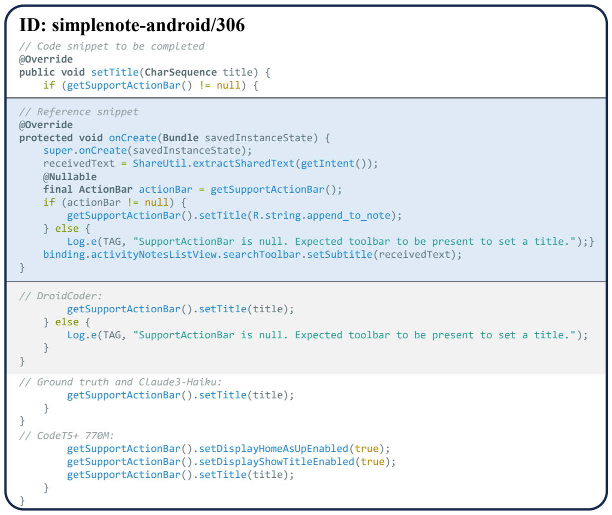
Considering the aforementioned impacts and the efficiency, we set *TopK* to 10.

Answer to RQ2: Each component in DroidCoder positively contributes to its code completion performance. Reranking and integrating functionally similar code snippets with context during fine-tuning are crucial. Functionally similar code snippets are more beneficial for code completion than contextual information.

5.3 RQ3: Completed Code Quality

The fundamental role of code completion is to accelerate the development process and assist developers in implementing requirements. Therefore, it is also essential to evaluate the quality of generated code from code completion approaches. Specifically, we assess the quality of the completed code by checking the following two conditions: First, the completed code snippet must pass compilation checks. Second, all method parameters must be used at least once. There are other conditions that can evaluate code quality, but we consider these two to be among the most fundamental. When both conditions are met, we consider the completed code to have passed our quality evaluation. Then, we define the **Positive Rate** as the percentage of all completed code that passes both checks and utilize this metric to measure the quality of completed code.

Table 5 shows the Positive Rate of different methods. From the table, we found that our method significantly improves code quality. Specifically, after fine-tuning with our method, the Positive Rate of original CodeT5+ 770M and StarCoderBase 1B increased by 214.42% and 350.63%, respectively. Notably, after fine-tuning with DroidCoder, the Positive Rate of these two models improved from being significantly lower than ChatGPT3.5 to surpassing it. This improvement may be because the model fully learns the syntax rules of different programming languages from closely relevant snippets and other contextual information during fine-tuning while also understanding other coding practices, such as the use of method parameters.



ID: simplenote-android/306

```java
// Code snippet to be completed
@Override
public void setTitle(CharSequence title) {
    if (getSupportActionBar() != null) {

// Reference snippet
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    receivedText = ShareUtil.extractSharedText(getIntent());
    @Nullable
    final ActionBar actionBar = getSupportActionBar();
    if (actionBar != null) {
        getSupportActionBar().setTitle(R.string.append_to_note);
    } else {
        Log.e(TAG, "SupportActionBar is null. Expected toolbar to be present to set a title.");}
    binding.activityNotesListView.searchToolbar.setSubtitle(receivedText);
}

// DroidCoder:
        getSupportActionBar().setTitle(title);
    } else {
        Log.e(TAG, "SupportActionBar is null. Expected toolbar to be present to set a title.");
    }
}
// Ground truth and Claude3-Haiku:
        getSupportActionBar().setTitle(title);
    }
}
// CodeT5+ 770M:
        getSupportActionBar().setDisplayHomeAsUpEnabled(true);
        getSupportActionBar().setDisplayShowTitleEnabled(true);
        getSupportActionBar().setTitle(title);
    }
}
```

Figure 8: Example for completed code by different methods.

Case Study. Figure 8 shows an example that includes the completion results of three methods: DroidCoder, Claude3-Haiku, and the original CodeT5+ 770M model. As shown in Figure 8, compared with the original output and the ground truth, our completion result not only completes successfully but also has an additional log, which comes from the similar implementation in other apps retrieved. Even the recent commercial generation model Claude3-Haiku just completes the essential code. Code completion should provide reliable code in addition to basic function implementations.

Human Study. Following the previous work [24], we also conducted human evaluation to inspect the code quality of DroidCoder's outputs. We invited six graduate students with over one year of Android development experience to participate in our human study. We randomly selected 60 samples in the test datasets and created an online questionnaire. For each sample, we included the code snippet to be completed and two completion outputs. Specifically, the participants were not informed whether these two completion outputs were generated by the original CodeT5+ 770M model or by DroidCoder, which also utilized the CodeT5+ 770M as its base model. They were asked to rate each completion output on a scale of 1 to 5, with 1 meaning completely unsatisfied and 5 meaning fully satisfying completion.

The questionnaire results show that DroidCoder's outputs achieved an average score of 4.39, while the original model's outputs only received 2.97. In all the samples, only 8.33% showed DroidCoder scoring lower than the original model, while in the remaining 91.67%, the scores of DroidCoder were either higher than or equal to the original model. This further demonstrates the effectiveness of DroidCoder in practical Android development.

The Fleiss' Kappa score [12] for this questionnaire is 0.61, indicating "substantial agreement" among the participants. Individual scoring preferences may have introduced some relative differences. For example, some participants tended to give lower scores, while others preferred to select mid-range values.

Answer to RQ3: DroidCoder can significantly enhance the quality of code completion compared to baseline models, surpassing existing methods. This indicates that DroidCoder is better suited for practical development, assisting developers more effectively.

6 THREATS TO VALIDITY

Internal Threats. The first internal threat relates to potential data leakage, which is common in recent works [33, 46]. To mitigate this threat, we ensure at least two Kotlin projects for each category of test apps since the datasets used to train the code LLMs we select do not include Kotlin projects. Additionally, the relatively poor performance of direct inference on pre-trained code LLMs can prove that there is little data leakage. Our effective results (RQ1) also illustrate consistent performance across various apps on different code LLMs.

Another possible internal threat is our choices of models and datasets. To mitigate it, we use similar criteria as in RepoCoder and choose more recent data whenever possible. Also, we select three widely used pre-trained models that share different architectures, specifically CodeGPT, CodeT5+, and StarCoder. We also plan to evaluate our work on larger code LLMs such as CodeLLaMa in the future. However, achieving better results on small-parameter models is still worth exploring. Our categories of apps and the choice of a 7:3 ratio for training and testing sets introduce a potential concern. Therefore, we experiment with another ten different apps on the same models and training set (Section 5.1), and the results illustrate our approach still achieves similar improvements.

External Threats. Our implementations and evaluations are specific to Android development. However, our insights and methodology can be transferred to other research problems. Specifically, our approach highlights the importance of selecting the most appropriate context based on the problem and exploiting features that facilitate retrieval and ranking to enhance RAG. This can significantly improve the performance of LLMs. Another important finding is that, traditionally, RAG relied solely on character similarity for ranking. But in many problems, ranking based on distinctive tokens can be more effective.

7 RELATED WORKS

Code completion has been continuously studied. Here, we focus on recent works, as they have shown superior potential.

7.1 Code LLMs for Code Completion

Code completion, as a key technology for enhancing efficiency, has been widely adopted in modern integrated development environments (IDEs) [6, 20, 52]. With significant advances in processing with large language models (LLMs) and pre-trained techniques [11, 27, 28], recent research has begun to explore their application to code completion [9], including Codex [57], CodeGen [34], CodeT5 [50], CodeT5+ [47], StarCoder [23] and CodeLlama [40]. These models treat code as sequences of tokens and markedly differ from previous works that utilize structured information such as abstract syntax tree (AST) [17, 20, 22, 26, 48]. Although these models have already achieved impressive success in conventional code completion [8], they still face challenges in specific domains or frameworks [35], especially the Android development environment. Firstly, neglecting the relationships between Android callbacks and UI widgets reduces code completion effectiveness. Secondly, multiple programming languages increase the difficulty of generalization and comprehension for the model. In general, improving the understanding of contextual information is still important to assist the code auto-completion task.

7.2 Repository-Level Code Completion

As software development scales up and code repositories become increasingly complex, contexts are no longer sufficient for code completion. Therefore, recent research has begun to explore how to integrate various information in the entire repositories to improve the code completion results. RepoBench [29] and CrossCodeEval [10] establish comprehensive benchmarks for evaluating the capabilities of the repository-level code completion. While ReACC [30] and RepoCoder [55] mainly focus on similar code snippets, GTNM [25], RLPG [41], Repoformer [51] and RepoHyper [35] investigate specific approaches for utilizing additional repository-level information. The main insight behind these approaches is capturing information from the same file and files in the same repository to enhance the pre-trained models. However, the Android framework has various unique features such as lifecycle callbacks and UI widgets, which are more difficult to utilize than the repository-level information such as the repository name and file names, thus understanding their informative relationships is crucial for Android code completion.

8 CONCLUSION AND FUTURE WORKS

In this paper, we propose a novel framework, DroidCoder, that leverages Android development features and contextual information of code snippets to enrich the RAG and integrates it during fine-tuning to enable the model to learn how to use the enriched RAG for more effective code completion. The experimental results demonstrate that DroidCoder can significantly improve the effectiveness of Code LLMs on code completion under Android development both at line-level and multi-line-level. DroidCoder also outperforms two state-of-the-art code completion tools based on retrieval-augmented language models under different completion scenarios and improves the quality of the completed code.

In the future, we will continue to focus on utilizing and integrating diverse information sources, such as git commits, which are short code snippets accompanied by descriptions that we believe can be useful for code completion. Additionally, we aim to transfer our approaches to various other software engineering tasks, particularly in industrial scenarios.

9 DATA AVAILABILITY

Our replication package[2] is publicly available.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. This research was supported by the National Natural Science Foundation of China under Grant Nos. 62372227 and 62032010.

[2]https://github.com/XR-Y/DroidCoder

REFERENCES

[1] 2019. Cretz/Kastree: Simple Kotlin Source AST and Syntax Parsing, Editing, and Writing. https://github.com/cretz/kastree.

[2] 2021. Seatgeek/Fuzzywuzzy. https://github.com/seatgeek/fuzzywuzzy.

[3] 2022. Kotlinx/Ast: Generic AST Parsing Library for Kotlin Multiplatform. https://github.com/kotlinx/ast.

[4] 2024. Introducing the next Generation of Claude. https://www.anthropic.com/news/claude-3-family.

[5] Akiko Aizawa. 2003. An Information-Theoretic Perspective of Tf–Idf Measures. *Information Processing & Management* 39, 1 (Jan. 2003), 45–65. https://doi.org/10.1016/S0306-4573(02)00021-3

[6] Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. Learning from Examples to Improve Code Completion Systems. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '09)*. Association for Computing Machinery, New York, NY, USA, 213–222. https://doi.org/10.1145/1595696.1595728

[7] Shaoheng Cao, Minxue Pan, Yu Pei, Wenhua Yang, Tian Zhang, Linzhang Wang, and Xuandong Li. 2024. Comprehensive Semantic Repair of Obsolete GUI Test Scripts for Mobile Applications. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 1096–1108.

[8] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2022. An Empirical Study on the Usage of Transformer Models for Code Completion. *IEEE Transactions on Software Engineering* 48, 12 (Dec. 2022), 4818–4837. https://doi.org/10.1109/TSE.2021.3128234

[9] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2021. An Empirical Study on the Usage of BERT Models for Code Completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 108–119. https://doi.org/10.1109/MSR52588.2021.00024

[10] Yangruibo Ding, Zijian Wang, Wasi 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. *Advances in Neural Information Processing Systems* 36 (Dec. 2023), 46701–46723.

[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[12] Joseph L Fleiss. 1971. Measuring nominal scale agreement among many raters. *Psychological bulletin* 76, 5 (1971), 378.

[13] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, Hongyu Zhang, and Michael R. Lyu. 2023. What Makes Good In-Context Demonstrations for Code Intelligence Tasks with LLMs?. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 761–773. https://doi.org/10.1109/ASE56229.2023.00109

[14] Qi Guo, Xiaohong Li, Xiaofei Xie, Shangqing Liu, Ze Tang, Ruitao Feng, Junjie Wang, Jidong Ge, and Lei Bu. 2024. FT2Ra: A Fine-Tuning-Inspired Approach to Retrieval-Augmented Code Completion. https://doi.org/10.1145/3650212.3652130 arXiv:2404.01554 [cs]

[15] R. Hosseini and P. Brusilovsky. 2013. JavaParser: A Fine-Grain Concept Indexing Tool for Java Problems. In *CEUR Workshop Proceedings*, Vol. 1009. University of Pittsburgh, 60–63.

[16] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. https://doi.org/10.48550/arXiv.2106.09685 arXiv:2106.09685 [cs]

[17] Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. 2022. CodeFill: Multi-Token Code Completion by Jointly Learning from Structure and Naming Sequences. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 401–412. https://doi.org/10.1145/3510003.3510172

[18] Paul Jaccard. 1912. THE DISTRIBUTION OF THE FLORA IN THE ALPINE ZONE.1. *New Phytologist* 11, 2 (Feb. 1912), 37–50. https://doi.org/10.1111/j.1469-8137.1912.tb05611.x

[19] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large Language Models Meet Program Synthesis. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1219–1231. https://doi.org/10.1145/3510003.3510203

[20] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code Prediction by Feeding Trees to Transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, Madrid, ES, 150–162. https://doi.org/10.1109/ICSE43902.2021.00026

[21] V. I. Levenshtein. 1966. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady* 10 (Feb. 1966), 707.

[22] Jian Li, Yue Wang, Michael R Lyu, and Irwin King. 2017. Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573* (2017).

[23] 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, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour 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. 2023. StarCoder: May the Source Be with You! https://doi.org/10.48550/arXiv.2305.06161 arXiv:2305.06161 [cs]

[24] Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Dong Chen, Shuai Wang, and Cuiyun Gao. 2023. CCTEST: Testing and Repairing Code Completion Systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1238–1250. https://doi.org/10.1109/ICSE48619.2023.00110

[25] Fang Liu, Ge Li, Zhiyi Fu, Shuai Lu, Yiyang Hao, and Zhi Jin. 2022. Learning to recommend method names with global context. In *Proceedings of the 44th International Conference on Software Engineering*. 1294–1306.

[26] Fang Liu, Ge Li, Bolin Wei, Xin Xia, Zhiyi Fu, and Zhi Jin. 2022. A Unified Multi-Task Learning Model for AST-level and Token-Level Code Completion. *Empirical Software Engineering* 27, 4 (April 2022), 91. https://doi.org/10.1007/s10664-022-10140-7

[27] Shangqing Liu, Yanzhou Li, Xiaofei Xie, and Yang Liu. 2022. Commitbart: A large pre-trained model for github commits. *arXiv preprint arXiv:2208.08100* (2022).

[28] Shangqing Liu, Bozhi Wu, Xiaofei Xie, Guozhu Meng, and Yang Liu. 2023. Contrabert: Enhancing code pre-trained models via contrastive learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2476–2487.

[29] Tianyang Liu, Canwen Xu, and Julian McAuley. 2023. RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems. https://doi.org/10.48550/arXiv.2306.03091 arXiv:2306.03091 [cs]

[30] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A Retrieval-Augmented Code Completion Framework. https://doi.org/10.48550/arXiv.2203.07722 arXiv:2203.07722 [cs]

[31] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin 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. https://doi.org/10.48550/arXiv.2102.04664 arXiv:2102.04664 [cs]

[32] Leonardo Mariani, Ali Mohebbi, Mauro Pezzè, and Valerio Terragni. 2021. Semantic Matching of GUI Events for Test Reuse: Are We There Yet?. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 177–190. https://doi.org/10.1145/3460319.3464827

[33] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2450–2462. https://doi.org/10.1109/ICSE48619.2023.00205

[34] 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. https://doi.org/10.48550/arXiv.2203.13474 arXiv:2203.13474 [cs]

[35] Huy N. Phan, Hoang N. Phan, Tien N. Nguyen, and Nghi D. Q. Bui. 2024. RepoHyper: Better Context Retrieval Is All You Need for Repository-Level Code Completion. https://doi.org/10.48550/arXiv.2403.06095 arXiv:2403.06095 [cs]

[36] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning Transferable Visual Models From Natural Language Supervision. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 8748–8763. http://proceedings.mlr.press/v139/radford21a.html

[37] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. 2022. Hierarchical Text-Conditional Image Generation with CLIP Latents. *CoRR* abs/2204.06125 (2022). https://doi.org/10.48550/ARXIV.2204.06125 arXiv:2204.06125

[38] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings Using Siamese BERT-Networks. https://doi.org/10.48550/arXiv.1908.10084 arXiv:1908.10084 [cs]

[39] Stephen Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Foundations and Trends® in Information Retrieval* 3, 4

(Dec. 2009), 333–389. https://doi.org/10.1561/1500000019

[40] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, 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. 2024. Code Llama: Open Foundation Models for Code. https://doi.org/10.48550/arXiv.2308.12950 arXiv:2308.12950 [cs]

[41] 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*. PMLR, 31693–31715.

[42] Ze Tang, Jidong Ge, Shangqing Liu, Tingwei Zhu, Tongtong Xu, Liguo Huang, and Bin Luo. 2023. Domain Adaptive Code Completion via Language Models and Decoupled Domain Databases. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Luxembourg, Luxembourg, 421–433. https://doi.org/10.1109/ASE56229.2023.00076

[43] 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* 28, 4 (Sept. 2019), 19:1–19:29. https://doi.org/10.1145/3340544

[44] Taha ValizadehAslani, Yiwen Shi, Jing Wang, Ping Ren, Yi Zhang, Meng Hu, Liang Zhao, and Hualou Liang. 2024. Two-stage fine-tuning with ChatGPT data augmentation for learning class-imbalanced data. *Neurocomputing* 592 (2024), 127801. https://doi.org/10.1016/j.neucom.2024.127801

[45] Mian Wan, Negarsadat Abolhassani, Ali Alotaibi, and William G. J. Halfond. 2019. An Empirical Study of UI Implementations in Android Applications. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 65–75. https://doi.org/10.1109/ICSME.2019.00016

[46] Chong Wang, Jian Zhang, Yebo Feng, Tianlin Li, Weisong Sun, Yang Liu, and Xin Peng. 2024. Teaching Code LLMs to Use Autocompletion Tools in Repository-Level Code Generation. https://doi.org/10.48550/arXiv.2401.06391 arXiv:2401.06391 [cs]

[47] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. https://doi.org/10.48550/arXiv.2305.07922 arXiv:2305.07922 [cs]

[48] Yanlin Wang and Hui Li. 2021. Code completion by modeling flattened abstract syntax trees as graphs. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 35. 14015–14023.

[49] Yihui Wang, Huaxiao Liu, Shanquan Gao, and Xiao Tang. 2023. Animation2API: API Recommendation for the Implementation of Android UI Animations. *IEEE Transactions on Software Engineering* 49, 9 (Sept. 2023), 4411–4428. https://doi.org/10.1109/TSE.2023.3294971

[50] 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. https://doi.org/10.48550/arXiv.2109.00859 arXiv:2109.00859 [cs]

[51] Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2024. Repoformer: Selective Retrieval for Repository-Level Code Completion. arXiv:2403.10059 [cs]

[52] Junwei Wu, Liwei Shen, Wunan Guo, and Wenyun Zhao. 2017. Code Recommendation for Android Development: How Does It Work and What Can Be Improved? *Science China Information Sciences* 60, 9 (July 2017), 092111. https://doi.org/10.1007/s11432-017-9058-0

[53] Yiheng Xiong, Mengqian Xu, Ting Su, Jingling Sun, Jue Wang, He Wen, Geguang Pu, Jifeng He, and Zhendong Su. 2023. An Empirical Study of Functional Bugs in Android Apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1319–1331. https://doi.org/10.1145/3597926.3598138

[54] Rui Yang, Lin Song, Yanwei Li, Sijie Zhao, Yixiao Ge, Xiu Li, and Ying Shan. 2023. GPT4Tools: Teaching Large Language Model to Use Tools via Self-instruction. *Advances in Neural Information Processing Systems* 36 (Dec. 2023), 71995–72007.

[55] 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. https://doi.org/10.48550/arXiv.2303.12570 arXiv:2303.12570 [cs]

[56] Yifan Zhang, Bryan Hooi, Dapeng Hu, Jian Liang, and Jiashi Feng. 2021. Unleashing the Power of Contrastive Self-Supervised Visual Models via Contrast-Regularized Fine-Tuning. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan (Eds.). 29848–29860. https://proceedings.neurips.cc/paper/2021/hash/fa14d4fe2f19414de3ebd9f63d5c0169-Abstract.html

[57] Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2023. A Survey on Language Models for Code. https://doi.org/10.48550/arXiv.2311.07989 arXiv:2311.07989 [cs]

[58] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Seenomaly: Vision-Based Linting of GUI Animation Effects against Design-Don't Guidelines. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, Seoul South Korea, 1286–1297. https://doi.org/10.1145/3377811.3380411