Repository-level Code Search with Neural Retrieval Methods

Siddharth Gandhi, Luyu Gao, Jamie Callan
Language Technologies Institute
Carnegie Mellon University
{ssg2,luyug,callan}@cs.cmu.edu

Abstract

Large Language Models (LLMs) excel at generating code in isolation but struggle with complex real-world scenarios involving multi-file dependencies and project contexts. This limits their use in code understanding and maintenance. Existing code search models are primarily focused on snippet-level searches, and not on repository-level searches that require understanding the context of past changes in a project. To address this, we propose a new system that combines information retrieval techniques like BM25 with neural reranking using CodeBERT. This system identifies relevant files in code repositories based on user queries or bugs. The goal is to enhance LLM's context understanding for better code completion by using snippets from these relevant files. We've achieved a 40% improvement in MAP/P@10 on test queries resembling real-world Github issues, using various reranking configurations with a BM25 baseline. Additionally, we're developing a dataset from over 80 popular Github repositories, including commit details, file edits, and diffs, to encourage further research in this direction. Our code and results obtained are available here[1].

1 Introduction

In the realm of software development, the advent of Large Language Models (LLMs) like GPT-4 (OpenAI, 2023) has disrupted the way developers approach coding and software maintenance. These models have shown remarkable proficiency in generating code snippets and providing coding assistance in limited contexts. Products like Github Copilot have been very successful commercially, with over 1 million paying customers (ZDNet, 2023).

However, LLM's utility is significantly hampered when it comes to dealing with real-world complexities. Addressing a bug often requires traversing through a vast repository, comprehending how functions interact across various files, or identifying a minor mistake within complex code. Thus, the ability to comprehend and manipulate code within the broader scope of an entire codebase remains a challenging frontier for LLMs, specifically due to their limited context window.

However, even as the context window of LLMs increases to 100K tokens (Anthropic, 2023), empirically we find that the quality of responses decreases while the risk of hallucination and overall cost increases (Liu et al., 2023a). Both research (Xu et al., 2023) and industry (Pinecone, 2023) sources find that a short, high-quality (retrieved) context performs significantly better than a longer, lower-quality context. Hence, well-tuned Information Retrieval (IR) methods like BM25 with robust reranking pipelines have the potential to help here, narrowing down the search scope immensely to provide better context to LLMs.

In the realm of source code search, current methodologies focus mainly on function or snippet-level queries, largely overlooking the nuanced demands of repository-level search. Repository-level means it encompasses a comprehensive view of the codebase, including the history and rationale behind particular changes (for instance, reflected in commit messages for code diffs), and the evolution of code across versions. This goes beyond just the source code, considering the interactions of files across time in the overarching software architecture. Such a detailed perspective might be key to understanding complex code dependencies and dynamics, making repository-level search crucial for effective software development and maintenance.

In this work, we adopt a multistage reranking approach for the task of repository-level code search. The current system is designed to pinpoint the most likely files to be fetched in response to user queries or bug reports, such as 'The context window breaks every time I click' or 'The contributor

[1] https://github.com/Siddharth-Gandhi/ds

gallery doesn't display the user's Github handle when active'. In future work, relevant portions from these files can be sent to a capable LLM like GPT4 or CodeLlama (Rozière et al., 2023), to recommend relevant code changes for such issues. We hypothesize this should lead to significantly better generations. Currently, we use both natural language (commit messages, file paths) and code (file contents, diffs) as relevance indicators to filter out relevant files for the given user query. Broadly, we start with a Github repository, index just the commit messages for the quick initial BM25 retrieval, and test various BERT reranking configurations on other relevance indicators to generate a final ranking. Finally, we evaluate our system on modified test queries by transforming past commit messages using an LLM, to mimic real Github issues.

2 Literature Review

2.1 Information Retrieval

Traditional methods like BM25 (Robertson et al., 2009) are fast and efficient, but fail to capture semantic meaning due to exact token matching. Recently, with the advent of Transformers (Vaswani et al., 2023), neural ranking models like BERT (Dai and Callan, 2019) or dense (Karpukhin et al., 2020) rerankers have emerged as state-of-the-art solutions, providing a more nuanced understanding of textual relevance. Multistage retrieval pipelines (Nogueira et al., 2019) often combine high recall rankers like BM25 with high precision rerankers (BERT) leading to significantly improved rankings.

2.2 Neural Code Models

BERT was only pretrained on Natural Language (NL). CodeBERT (Feng et al., 2020), tries to employ bimodal pre-training with both programming languages (PL) and NL, with Masked Language Modelling (MLM) and Replaced Token Detection (RTD) objectives, leading to significant improvements in tasks like natural language code search and code documentation generation. GraphCodeBERT (Guo et al., 2021) further advances this by incorporating the inherent structure of code, particularly data flow, into its pre-training, leading to superior performance in code search, clone detection, code translation, and refinement. Finally, with the advent of LLMs trained on Next Token Prediction, systems like CodeT5 (Wang et al., 2021), StarCoder (Li et al., 2023) and CodeLlama specializes in understanding and generating code, leveraging code semantics and developer-assigned identifiers for enhanced performance in tasks like code generation.

2.3 Repo-Level Code Tasks

Repository-level code completion has seen significant advancements through innovative frameworks. RepoCoder (Zhang et al., 2023) enhances code completion by over 10% using a sparse similarity-based retriever and various pre-trained code LLMs. CCFINDER (Ding et al., 2023) proposes to integrate cross-file context into pre-trained code LLMs by building a project context graph, resulting in a 28.69% increase in identifier matching. The Repo-Level Prompt Generator (Shrivastava et al., 2023) proposes generating context-specific prompts with an oracle retriever, achieving improvements over Codex for single-line code completions. Finally, SWE-Bench (Jimenez et al., 2023) tries to solve real Github issues directly using BM25 on code files, and passes as much context as available in the LLM's context that they evaluate (max 32K). They could only solve 4.8% issues with this approach.

2.4 Neural Code Search & Benchmarks

The semantic code search task defined by popular code benchmarks like CodeSearchNet (Husain et al., 2020) or CodeXGLUE (Lu et al., 2021) involves retrieving relevant code snippets for a natural language query from approximately 6 million functions across six programming languages. Models like CodeBERT/GraphCodeBERT form queries from a function's documentation, and use a BERT reranking by encoding both query and code, and computing relevance scores using `[CLS]` token. Gu et al. (2018) tries doing this by using dense reranking, embedding both query and code in higher-dimensional space, and doing a similarity search.

This, however, is not quite relevant to the repository-level code file search task that we have which requires analyzing entire repositories. Unlike snippet-level search, repository-level search necessitates a more intricate understanding of a larger and more complex codebase, where the focus extends beyond individual functions to include the interconnections and dependencies among various files, with the context of changes over time. This wider scope aims to capture the essence of a software project as a whole, addressing the needs of comprehensive code understanding and maintenance in real-world development scenarios.

In our limited search, we were not able to find any papers which tried to do such code file search. The closest system attempting such a task was Github in its recent GitHub Universe 2023 demo (GitHub Next, 2023).

The closest benchmarks we could find were SWEBench (Jimenez et al., 2023) proposing a repository-wide completion task, with 2294 Github Issues across 12 Python repositories, and RepoBench (Liu et al., 2023b) where they develop repository-level retrieval benchmark with 12,000 samples for Python and Java. However, both of them are for single/few line completions, not code search specifically.

Hence, we find two big gaps in the current literature:

1. Lack of focus on repository-level code file search. While existing literature extensively covers code snippet search and repository-level code completion, there is a notable gap in research focusing on repository-level code file search (finding relevant files to be fetched for a given user query). This is an important task which can be used for richer context for LLMs downstream for better completion or understanding of tasks.

2. Lack of high quality, multi-language, code file search + snippet completion benchmark or datasets at the repository level. Most of the current benchmarks focus on function-level searching tasks (given a natural language query, which one of the 6M functions is most relevant).

3 Proposed System

Our system aims to take some form of query as input and gives a list of potential files as output. And since this is at the repository level, that is each repository will have its own dataset, ranker and rerankers, we repeat this approach across 10 different repositories and report macro-results. We discuss the important aspects of our system below.

3.1 Dataset

There were not any pre-existing datasets available for the task of repository-level code-file search. Hence, we create our dataset from scratch, with the following objectives in mind:

- Capturing a wide range of programming languages and software projects, ensuring diversity in terms of size, complexity, and domain.

- Including detailed commit histories, allowing for a deep understanding of the evolution of the codebase and the context behind changes.

- Storing all versions of a file, along with its respective commit IDs, messages, diffs and status (according to Github API convention namely, `modified/added/deleted/renamed`).

We start by querying the GitHub API to find the most starred repositories of all time. We then filter the non-code related repositories (such as `EbookFoundation/free-programming-books` having 300K+ stars, but is just a collection of useful links, not a software) using the `language` attribute from API response matching against a list of all major programming languages. Still, we found many non-software repositories remaining, presumably because of having a `language` attribute even with one code file. So, we manually select about 100 most popular GitHub software repositories, such as `react`, `pytorch`, `angular`, `kafka`, etc.

Next, we scrape the contents of these repositories to extract relevant information mentioned in the objectives. We first tried using Github API, but that quickly became infeasible given the millions of API calls needed. Instead, we clone each repository and use git commands locally to query for the given information (e.g., `git show --name-only <commit-hash>` to get a list of files modified for a given commit, or `git show <commit-hash>:<file-path>` to get a version of the file at a particular commit). We also filter non-code files such as READMEs or config files, by matching against a programming extensions list[2].

We utilize GitPython[3] to do the scrape efficiently, which provides optimized well-tested abstractions in Python for git commands with minimum process spawns. It took approximately 2 full days on 4 compute nodes with SSDs to finish the scraping with approximately 150 GB of data (including BM25 index, expanded in Sec 3.4). Data for each repository is stored in multiple parquet files, because of its efficient compression, fast reads and cross-platform compatibility. A sample row is shown in Table 1.

[2] https://github.com/Siddharth-Gandhi/ds/blob/main/misc/code_extensions.json

[3] https://github.com/gitpython-developers/GitPython

Column	Sample
owner	facebook
repo_name	react
commit_date	1575406296
commit_id	f523b2e0d369e3f42938b56784f9ce1990838753
commit_message	Use fewer global variables in Hooks (#17480)...
file_path	packages/react-reconciler/src/ReactFiberThrow.js
previous_commit_id	d75323f65d0f263dd4b0c15cebe987cccf822783
previous_file_content	@flow\n */\n\nimport type {Fiber} from \'./ReactFiber\';\nimport...
cur_file_content	@flow\n */\n\nimport type {Fiber} from \'./ReactFiber\';\nimport...
diff	@@ -195,6 +195,18 @@ function throwException(\n...
status	modified
is_merge	False
file_extension	js

Table 1: Format of data stored in parquet file - `previous_*` are set when `status` is not `added`, `is_merge` is `True` when `commit_id` has > 2 parents (merge of 2 branches), `date` is in UNIX format.

3.2 Task Definition

We define the task of repository-level code search as follows: given a user query q, find a list of files F which are most likely to address the question or bug raised in q, given the previous commit history of the repository. Importantly, we want to retrieve a file list, however, the dataset we have is at the commit level, with one file edited per row, and that same file can be edited across multiple commits, either before or after that commit's timestamp. Hence, two important points arise 1. while training on past commit history, we must mask out future versions of the files to prevent leakage and 2. thinking about which version of files to use for reranking (explored more in Sec 3.5 and 3.6).

3.3 Evaluation Metrics

For this task, we use usual IR metrics to evaluate our system: Mean Average Precision (MAP), Precision at k (P@10, P@100, P@1000) Mean Reciprocal Rank (MRR), Recall at k (Recall@100, Recall@1000). More specifically, given a test query t_q for a commit t_c, a list of actual modified files F_a for t_c, our system predicted list of files F_p, and a relevance list R where $R[i] = 1$ if file $F_p[i]$ is in F_a, we have:

- $P@k = \frac{1}{k} \sum_{i=1}^{k} R[i]$
- $MRR = 1/(j+1)$ where j is the index of first relevant file in F_p (if none, then 0)
- $Recall@K = \frac{1}{|F_a|} \sum_{i=1}^{k} R[i]$
- Average Precision $(AP) = \frac{\sum_{j \in \mathcal{X}} P@j}{\sum_{i=1}^{k} R[i]}$, $\mathcal{X}$ is list of indices i where $R[i] = 1$ and macro-averaged across $|Q|$ test queries, $MAP = \frac{AP}{|Q|}$.

We do not use NDCG (Normalized Discounted Cumulative Gain), because we do not have multi-graded relevance labels for our test queries. For a given predicted file f, and actual modified list F_a, we only know whether f was present in F_a, not how important f was relative to other files in F_a.

3.4 BM25 Baseline

We select BM25 (Robertson et al., 2009) as a baseline and initial ranker for further rerankers because of its efficiency and high recall over a variety of datasets. We use PySerini's (Lin et al., 2021) implementation of BM25 in Python.

We use BM25 to match user queries against documents (like rows in Table 1) to score each commit message (since multiple rows can have the same commit message for different edited files). However, our task is to retrieve a set of files, not commits. So we need to aggregate BM25 results across the files.

File Based Aggregation Say for a given query q, BM25 retrieves commits $[c_1, c_2, c_3]$ where c_1 had edited files $[f_1, f_2, f_3]$ and received a BM25 score 5, $c2$ had files $[f_2, f_5, f_1, f_6]$ with score 3, and $c3$ had $[f_4, f_3, f_2, f_7, f_8]$ with score 1. Then we can aggregate scores across files as $[f_2 : 9, f_1 : 8, f_3 : 6...]$ using various aggregation strategies like `sump` (shown), `firstp` or `avgp`. In our preliminary experiments, we found that `sump` performed the best (in Recall@100), so we use that for the remaining experiments.

Masking Since we use BM25 to train reranking models on previous commit history in later sections, we also do masking of commits using the `commit_date` attribute. This ensures that if we were training on past versions of a file, the future ver-

sions are not leaked at any point. At inference time, no masking would be required since there would not be future versions of the files.

Tokenization PySerini has its own lexical tokenizers, but those are for natural language. Queries like Github issues or user bugs (often by developers) are likely to have programming snippets as part of the query, where such tokenizers might be suboptimal. Hence, we use TikToken (OpenAI, 2023), which is a fast Byte Pair Encoding (BPE) tokenizer trained on both code and natural language. We first tokenize the `commit_messages` and then use PySertini (with `-pretokenized` flag) to build an individual commit message index for each repository.

3.5 BERT Reranker

BM25 matches commit messages with exact-token matches and does not capture the semantic meaning and context of each message. Thus to improve the rankings, we use a BERT reranker (Dai and Callan, 2019), on the commit messages to rerank the top k documents (files in this case). We use a BERT model trained on code, such as `microsoft/codebert-base`. We embed the query and commit messages (passage) together (separated by `[SEP]` token) using the BERT model, and get relevance scores by passing the `[CLS]` token representation through a linear layer. This score captures the semantic similarity between the query and the passage.

However, for each file f, it can have many potential commits $C = [c_1, c_2, c_3]$ in which it was edited, so a question arises - which commit messages should we choose? We follow a simple idea - because of the way we do file aggregation in BM25, the commit list C will always be sorted by their initial BM25 scores. So we just pick a fixed number of these commits (hyperparameter) and aggregate their results to get one score per file. This is very similar to the passage count hyperparameter in the usual BERT reranking settings.

Finetuning BERT CodeBERT by default is general purpose and not designed for any specific task such as reranking. We experimented without finetuning, and the results were random, significantly worse than just using BM25. Hence, we finetune CodeBERT by using the Mean Squared Error (MSE) loss between the prediction score and label on (query, passage, label) triplets, constructed from past training data. Here, MSE is used because our model with linear layer head predicts a real-valued score between 0-1, not just binary 0/1 and, the triplets are constructed as follows: for a set of training queries T_q,

- **Query:** A query $q \in T_q$, where q[4] is either a commit message directly from training dataset (similar to `commit_message` from Table 1) or some modification of it, such as using a LLM like GPT4 to transform such commit messages into potential queries or issues which might have resulted in the commit. This is done to learn potentially better models grounded on pseudo-realistic data, however, we also hypothesize that just vanilla commit messages should also be good enough for training.

- **Passages:** For a given query q, we retrieve BM25 results $R_{bm25} = [c_1, c_2, ...]$ at the commit level (i.e. not file aggregated), since BERT only cares about query-commit message similarity and does not consider files. Then for each commit c[4] in R_{bm25}, c's commit message is a potential passage. From Appendix A.2, commit messages are short and can generally fit within the 512 sequence length of BERT (worst case, 50 tokens might be truncated), so we do not need to split each passage into subpassages. However, there could be hundreds of retrieved commits for q, so we first sort R_{bm25}, by the number of common files between q and c.

- **Label:** Next, we pick first N_+ (hyperparameter) non-0 scored passages (c'commit message) from the sorted R_{bm25} and assign them label 1 (implying that q and c are related in some way, since they changed similar files) and next N_- (hyperparameter) with label 0 (q and c's commit message are not related since they do not change similar files).

The goal is for (passage) commits with similar edited files to train (query) commits to get a higher score using our model. This would imply understanding the underlying motivation behind the changes and being able to retrieve those same files, should a similar query occur.

[4]Here commits c or queries q are objects similar to Table 1, and we can fetch individual commit message or list of files edited in that particular commit easily from the dataset

Repository	Commit Queries Avg Tokens	Short Queries Avg Tokens	Avg Files Edited / Query	Median Files Edited / Query
facebook_react	180.5	36.4	5.1	4
angular_angular	164.6	37.5	3.5	2
apache_spark	576.8	38.4	3.9	3
apache_kafka	198.3	39.7	5.4	3
django_django	52	25.3	2	2
julialang_julia	84.6	31.4	2.4	2
ruby_ruby	121.5	33.1	1.4	1
pytorch_pytorch	326.6	36.8	3.4	3
huggingface_transformers	231.2	33	5.1	4
redis_redis	161	36.3	2.4	2
Average	209.7	34.8	3.5	2.6

Table 2: Test Query Set Statistics Per Repository (Tokenized using `microsoft/codebert-base` tokenizer)

Combined BERT We prepare the triplet data and train individual BERT reranker models for each repository. But an interesting question arises - what happens if we combine all data across 10 repositories and train one big model? We hypothesize that it might not provide very good relevance indicators since different repositories have different commit message contents, conventions or files edited, so BERT might not learn the best relevance indicators from such diverse data.

3.6 Code Reranker

So far, we only utilize commit messages to do our rankings but not the code files themselves. Here, we develop an initial model for integrating relevance indicators from the code files. We again finetune a `microsoft/codebert-base` model, using the same loss as 3.5, and we embed `[SEP]` separated query and `(file_path + ' ' + code_snippet)` to get relevance scores. Training triplets are also constructed differently where for a set of training queries T_q,

- **Query:** A query $q \in T_q$, remains the same from 3.5
- **Passage:** For q, we retrieve BM25 results $R_{bm25} = [f_1, f_2, ..]$ but file-aggregated this time. Since we want to match query to code, we also need to decide which version of each file to focus on. Currently, we use the most recent version of the file.
- **Label:** If f was edited in q^4, we set label as 1 otherwise 0. We get N_+ positive files and N_- negative files.

Splitting into subpassages Unlike commit messages, which are a few hundred tokens long, code files are a few thousand tokens long. So we need to split them into subpassages (code snippets) to fit them in BERT's sequence length of 512. Here, subpassage length, count and stride are all hyperparameters. Unlike natural language passage splitting, we cannot split on whitespace, so we split in the tokenized version of the code. Even still we can have potentially 50 or 100 passages per code file, and it would be inefficient to compute scores for all. So which passages to focus on? Currently, we filter subpassages by the matching number of common lines between each subpassage and the `diff` of that file from the same commit c^4. The same labels as the original passage are assigned to all (query, subpassage) pairs.

3.7 Experimental Setup

We experiment with 10 repositories, with individual repository details shown in Appendix A.1. A potential concern was the presence of 'easy files', which are files edited so often across commits that it would be trivial for our system to rank them at the top, thus inflating metrics. We looked at the median edit frequencies and found that the most edited files are still only in $< 2\%$ of commits on average. Thus a Zipfian edit distribution is not likely, and metrics inflation should not be a problem.

Query Sets We have two types of query set: 1. **Commit**: Commit messages directly from the training data (similar to Table 1), generally being around 200 tokens long, and 2. **Short:** Modified query set, by passing the commit messages to GPT-4 and asking it to generate the most likely problem in the style of a Github issue for the given commit in 1-2 sentences, while masking the solution presented in the commit message. The full prompt along with example transformations for sample queries

is available in Appendix A.3. This is generally $30 - 40$ tokens long and is done to mimic real-world settings where developers generally describe an issue in a few sentences without knowing too many details about the problem and its solution. We also experimented with GPT-3.5 Turbo, however, we found that there was significant leakage, that is, the modified query would contain the solution, thereby defeating the purpose of this masking.

Thus, we have 2 query sets Commit and Short, for both training and testing. In each of the 10 repositories, for the commit train set, we have 1000 random training queries from the history, while for (GPT-modified) short train set, we have 500 curated queries (filtering outlier commits with too many files edited or too big commit messages). For testing, there are also 100 curated test queries, which are the same for both commit and test sets with the only difference being if the query is the vanilla commit message or transformed. Important statistics for the test set are available in Table 2.

BM25 We use default parameters provided in PySerini for initial ranking ($k1 = 0.9, b = 0.4$). For BM25 file aggregation, we use `sump` in all our experiments to maximize Recall@100. We retrieve at most $k = 1000$ initial files for initial retrieval, however because of masking future files during training, the candidate list size decreases to around 700.

For both BERT and CodeBERT We train these models on Nvidia Quadro RTX6000 (25 GB VRAM) with a batch size 32, for 10 epochs, with a learning rate of $5e - 5$, train depth of 1000 (number of BM25 results retrieved per train query), `sump` aggregation strategy, and $N_+ = N_- = 10$. Each repository's reranker is trained with either 1000 (Commit) or 500 (Short) train queries depending on the train query set.

BERT Reranker We set (commit) passage count to 5, with rerank depth of 250. For triplet label distribution, in commit train queries (1000 total) it was $(+, -) = (5602, 10567)$, while in short train queries (500 total) it was $(+, -) = (2489, 4948)$. Label imbalance is presumably because of missing relevant files in top k due to BM25's still mediocre Recall@1000 (see Sec 4). It takes about 45 minutes on average to train.

Combined BERT Here we just merge all training data for 10 repositories into one model. So this will have $500 * 10 = 5000$ short train queries and $1000 * 10 = 10000$ commit train queries. It takes about 6 hours to train (on commit set).

CodeReranker We set subpassage count to 25 (from the code file), with each subpassage length as 350 and a stride of 250. Rerank depth is set to 100 (generally followed after BERT Reranker). For label distribution, we were only able to train on the short train query set and we had $(+, -) = (25367, 123362)$. This is very large because we are breaking each file into subpassages and for each positive or negative code file, we add subpassage count (25 in our case) instances. It takes about 3 hours on average to train on the short query set (yet to explore training on the commit set). For experiments, we report results by reranking the top 100 results from the Combined BERT model trained on the commit set (since it had the highest Recall @ 100).

4 Results & Discussion

This section studies results listed in Table 3, often citing results using the indices (*a*-*l*) from the table. An important metric to keep in mind from Table 2 is, that there are approximately 3 files edited per test query. Since the goal is to rank these as high as possible, the most important metrics are P@10 or MRR.

BM25 performs reasonably well *(a)*, with the only important metric being Recall, since it will be used as the initial ranker. Recall@1000 is decent, around 0.68 meaning about 2/3 of relevant files are being retrieved in the top 1000 files on average. This implies there is a lot of scope to improve, potentially by tuning BM25 parameters.

Perhaps a larger value of $k1$ (term frequency scaling) might be useful to not saturate scores too quickly in case a lot of the commit messages have many overlapping terms due to conventions (here scores will be similar and thus rankings will be sub-optimal). Slightly higher Recall@1000 for Short queries *(g)*, where this might be less likely since due to being modified by an LLM, probably verifies this. It is more difficult to judge the effect of b (document length normalization). Given that we want to retrieve files, it is hard to see how that will be related to the length of the commit messages. Thus, tuning b is probably best left to trial and error. Finally, the massive drop in MAP in short test queries *($a \rightarrow g$)* is probably explained due to the

	Model	Train Query Set	Test Query Set	MAP	P@10	P@100	P@1000	MRR	R@100	R@1000
(a)	**BM25**	-	Commit	0.196	0.073	0.016	0.002	0.258	0.533	0.687
(b)	**BERT Rerank @ 250**	Commit	Commit	0.278	0.099	0.018	0.002	0.358	0.596	0.687
(c)	**BERT Rerank @ 250**	Short	Commit	0.213	0.085	0.017	0.002	0.280	0.581	0.687
(d)	**Combined BERT @ 250**	Commit	Commit	0.308	0.105	0.018	0.002	0.380	0.603	0.687
(e)	**Combined BERT @ 250**	Short	Commit	0.240	0.088	0.017	0.002	0.312	0.587	0.687
(f)	**Combined BERT @ 250 + CodeReranker @ 100**	Short	Commit	0.149	0.061	0.018	0.002	0.203	0.603	0.687
(g)	**BM25**	-	Short	0.196	0.070	0.015	0.002	0.263	0.521	0.689
(h)	**BERT Rerank @ 250**	Commit	Short	0.210	0.078	0.016	0.002	0.274	0.560	0.689
(i)	**BERT Rerank @ 250**	Short	Short	0.230	0.086	0.017	0.002	0.295	0.582	0.689
(j)	**Combined BERT @ 250**	Commit	Short	0.281	0.099	0.018	0.002	0.352	0.590	0.689
(k)	**Combined BERT @ 250**	Short	Short	0.281	0.102	0.017	0.002	0.354	0.589	0.689
(l)	**Combined BERT @ 250 + CodeReranker @ 100**	Short	Short	0.179	0.064	0.018	0.002	0.242	0.590	0.689

Table 3: Test Set Evaluation for Different Configurations

exact-token match of BM25, with LLMs replacing many relevant tokens from commit messages and introducing stochasticity.

BERT BERT gives a sizeable boost in metrics (15-30%), which is expected from better evaluating the semantics of commit messages. Interestingly, BERT when trained on commits performs very well on commit test set *(b)* (expected), but is only marginally better than BM25 on short test set *(h)*, implying some amount of overfitting perhaps. On short train queries *(c & i)*, it is vice-versa, however, the differences between short and commit test sets are more modest, implying better generalization. Also, we see drops in average performance on short test queries, presumably meaning the query modifications make it a hard task for neural systems as well. Finally, it takes about 5s/query which makes it pretty efficient.

Combined BERT This is the most significant result with about 40% improvements in MAP and MRR scores, compared to BM25. MRR is close to $1/3$ implying we are putting one relevant file in the third position already on average. This is also perhaps expected since this has a lot more training data (10x) compared to individual rerankers for each repository. More surprising however is that this approach works at all. We hypothesized that due to vastly different training data, BERT would not be able to understand the nuances of each repository. But clearly, it can benefit from the diverse source of data and can discriminate even better when focusing on individual repositories (similar idea to why pre-training is so successful).

CodeReranker We see CodeReranker significantly under performing *(f & l)*, even worse than just BM25. Apart from correctness issues, a primary reason might be that the Combined BERT model before CodeReranker is trained on the commit query set, while the CodeReranker itself is trained on the short query set (a mistake in configuration on our end). Perhaps this difference is confusing the model. For just one repository (`facebook_react`), we verified Combined BERT (short query train) followed by CodeReranker (also short query train), and we see minor improvements (around 5%) over the best Combined BERT model.

However, the counterargument can also be made that it is unlikely how rankings from the previous reranker, should affect the current reranker given the difference in Recall@100 between the two is quite small *((d vs e) or (j vs k))*. Perhaps the code subpassages being fed to CodeReranker are not good relevance indicators and better selection strategies be explored. We rule out code snippets being cut out due to the max sequence length of BERT, since this is being trained on short queries, and from Appendix A.2, we have $35 + 17 = 52$ tokens for query and file path combined. This leaves 460 tokens for code, thus no truncation happens.

Perhaps, just more code subpassages per code file are needed. On average we saw around 60-70 subpassages per code file, however, we only utilize 25 of them currently, meaning we are losing a lot of context from not having enough subpassages. Finally, it takes around 35s/query, so this is fairly inefficient - probably because of computing scores for multiple code subpassages.

Comparison between Train Query Sets We notice that Combined BERT trained on commits or short queries *(j & k)* perform pretty much the same

on the short test set (there is a drastic difference on commit test set *(d & e)*, but short test set is a more realistic scenario). This is fascinating, as it implies that training on LLM-modified queries did not help generalizability significantly. Even just in individual training scenarios *(h & i)*, the difference is minor. This probably implies that even though the model somewhat overfits to commit data (*d* being an outlier), it still learns very general patterns from the commit messages. This can help us save the cost the generating the GPT modified queries while not sacrificing any performance by training on just more number of past commits.

Potential Anomaly We notice that MRR for our results is generally quite high, around 0.25 in the worst case with BM25. This means, on average, a relevant file appears in the 4th position in the ranking. However, that should also mean we should see a P@10 of about 0.1 on average, but currently, it is lower than that (about 0.07).

This is possible in scenarios where 1. for some queries, relevant documents are placed very highly (maybe first or second rank), thus inflating the overall MRR, while 2. for other queries, there are no relevant documents in the top 10 at all (maybe in the top 20), leading to 0 P@10 scores but non-0 MRR scores. Such scenarios could inflate the MRR scores (since MRR=1 for one test query can compensate for MRR=0.1 for 10 other queries), while P@10 stays low consistently. Overall, this highlights that the rankings are quite inconsistent, where for some queries our system performs extremely well, while in others it performs quite poorly.

The only exceptions are Combined BERT models *(d & k)* which have MRR $\geq 1/3$ and P@10 = 0.1, implying both metrics agree. This means that the rankings are quite consistent and the system performs well for all test queries on average. This further highlights the extreme generalizability of the combined training approach.

5 Conclusion

In summary, our project combines traditional information retrieval methods with neural code models to develop a multi-stage reranking pipeline for the task of repository-level code search. This involves understanding the rationale behind previous changes to files from past commit history. We used BM25 and (Code)BERT reranking, utilizing a variety of relevance indicators such as commit messages, file paths and code contents, to fetch potential files to be edited in response to a user query (such as a bug or issue). Our tests, which included a diverse set of GitHub repositories and modified queries to represent real GitHub issues, showed that our approach is quite effective. We observe that models trained on combined data across multiple repositories perform best in this task, leading to a 40% improvement in metrics over the BM25 baseline. In the future, we hope this work helps to develop better LLM tools for understanding and maintaining complex, multi-file projects.

6 Future Work

There are multiple directions to improve this work, some of which are listed below.

- The most promising approach is Combined BERT, which seems very generalizable. So scaling this system up with more train queries and more repositories to get an even more diverse source of data is promising.
- The underwhelming utilization of code as a relevance indicator is apparent in the poor performance of CodeReranker. Better techniques could be explored to utilize code files efficiently. Perhaps BERT Rerank is a poor choice for code files due to their large sizes, and thus dense embeddings/reranker for code files is a better approach.
- Multi-relevance training to improve the performance of CodeReranker - instead of just assigning binary scores to each code subpassage based on whether it is part of a diff or not, we instead assign scores by the number of common lines with the diff. This might give better relevance indicators to train a better model, however, efficiency will still be an issue.
- A better sweep of hyperparameters such as BM25 parameters, rerank depths, optimal passage count, length, and strides is always a good idea.

References

Anthropic. 2023. Introducing 100k context windows.

Zhuyun Dai and Jamie Callan. 2019. Deeper text understanding for ir with contextual neural language modeling. In *Proceedings of the 42nd international ACM SIGIR conference on research and development in information retrieval*, pages 985–988.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023. Cocomic: Code completion by jointly modeling in-file and cross-file context.

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.

GitHub Next. 2023. Copilot workspace.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. ICSE '18, page 933–944, New York, NY, USA. Association for Computing Machinery.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcode{bert}: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. Codesearchnet challenge: Evaluating the state of semantic code search.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues?

Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. *arXiv preprint arXiv:2004.04906*.

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!

Jimmy Lin, Xueguang Ma, Sheng-Chieh Lin, Jheng-Hong Yang, Ronak Pradeep, and Rodrigo Nogueira. 2021. Pyserini: A Python toolkit for reproducible information retrieval research with sparse and dense representations. In *Proceedings of the 44th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2021)*, pages 2356–2362.

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

Tianyang Liu, Canwen Xu, and Julian McAuley. 2023b. Repobench: Benchmarking repository-level code auto-completion systems.

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.

Rodrigo Nogueira, Wei Yang, Kyunghyun Cho, and Jimmy Lin. 2019. Multi-stage document ranking with bert. *arXiv preprint arXiv:1910.14424*.

OpenAI. 2023. Gpt-4 technical report.

OpenAI. 2023. Tiktoken github.

Pinecone. 2023. Less is more: Why use retrieval instead of larger context windows.

Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389.

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

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention is all you need.

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.

Peng Xu, Wei Ping, Xianchao Wu, Lawrence McAfee, Chen Zhu, Zihan Liu, Sandeep Subramanian, Evelina Bakhturina, Mohammad Shoeybi, and Bryan Catanzaro. 2023. Retrieval meets long context large language models.

ZDNet. 2023. Microsoft upgrades copilot with openai's gpt-4, turbo dall-e 3, and more.

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.

A Appendix

A.1 Overall Repository Statistics

Table 4 highlights repository-level statistics like files edited and edit frequency of files.

A.2 Train Query Set Statistics for CodeReranker

Table 5 highlights average tokens for each files, queries and remaining tokens for code.

A.3 Query Modification Prompt

The following prompt was provided to GPT4 to modify commit messages into short queries:

```
You are a professional software developer who is
     given a very specific task. You will be
    given commit messages solving one or more
    problems from any big open-source Github
    repository, and your task is to identify
    those core problem(s) that this particular
    commit is trying to solve. With that
    information, you need to write a short
    description of problem itself in the style
    of a Github issue which would have existed *
    before* this change was committed. In
    essence you are trying to reconstruct what
    potential bugs led to certain commits
    happening. Do not mention any information
    about the solution in the commit (as that
    would be cheating). Just what the potential
    issue or problem could have been that led to
     this commit being needed.
Do not mention the names or contact information
    of any people involved in the commits (like
    authors or reviewers). Do not start
    responding with any description or titles,
    just start the issue. Limit your response to
     at most 2 lines. Just think of yourself as
    a developer who encountered a bug in a hurry
     and has to write 2 lines which captures as
    much information about the problem as
    possible. And remember do NOT leak any
    details of the solution in the issue message
    .

For example, given the commit message below:
'[Fizz] Fix for failing id overwrites for
    postpone (#27684)
When we postpone during a render we inject a new
     segment synchronously which we postpone.
    That gets assigned an ID so we can refer to
    it immediately in the postponed state.

When we do that, the parent segment may complete
     later even though it's also synchronous. If
     that ends up not having any content in it,
    it'll inline into the child and that will
    override the child's segment id which is not
     correct since it was already assigned one.

To fix this, we simply opt-out of the
    optimization in that case which is
    unfortunate because we'll generate many more
     unnecessary empty segments. So we should
    come up with a new strategy for segment id
    assignment but this fixes the bug.

Co-authored-by: Josh Story <story@hey.com>'

I want you to respond similar to:
'Synchronous render with postponed segments
    results in incorrect segment ID overrides,
    causing empty segments to be generated
    unnecessarily.'
```

The modified queries for all repositories are available on our GitHub project[5]. In total, it costs $120 in OpenAI API credits to modify around 6000 queries ((500 train queries + 100 test queries) * 10 repositories). Table 6 contains examples of transformed queries for one sample commit from `facebook_react`, `angular_angular` and `apache_kafka` respectively.

[5] https://github.com/Siddharth-Gandhi/ds/tree/main/gold

Repository	Total Commits	Total Files Edited	Mean Edit Freq Per File	Median Edit Freq Per File	Mean Files Edited Per Commit	Median Files Edited Per Commit
facebook_react	11609	73765	10.1	4	6.4	2
angular_angular	19464	151904	7.1	3	7.8	2
apache_spark	33679	188006	13.3	4	5.6	2
apache_kafka	10445	75655	9.9	4	7.2	2
django_django	21991	81252	18.3	7	3.7	2
julialang_julia	46778	182112	41.7	8	3.9	2
ruby_ruby	70211	180467	10.1	3	2.6	1
pytorch_pytorch	59554	276846	14.1	6	4.6	2
huggingface_ transformers	11157	56363	13.3	6	5.1	1
redis_redis	11077	29533	15.9	2	2.7	1
Average	29596.5	129590.3	15.38	4.7	4.96	1.7

Table 4: Overall Statistics Per Repository

Repository	Commit Train Queries Avg Tokens	Short Train Queries Avg Tokens	File Path Avg Tokens
facebook_react	242.1	36.1	20.9
angular_angular	197.1	36.7	20.3
apache_spark	819.9	39.6	30.3
apache_kafka	254.9	37.1	28.1
django_django	59.9	27.3	14
julialang_julia	124.3	32.2	9
ruby_ruby	156.1	32	6.6
pytorch_pytorch	494.5	38.4	14.5
huggingface_ transformers	388.6	33.8	15.2
redis_redis	236.1	37.5	8.1
Average	297.4	35	16.7

Table 5: Overall Token Statistics for training sets per repository (tokenized by `microsoft/codebert-base`)

Original Commit Message	GPT-4 Transformed Short Query
Extract queueing logic into shared functions (\#22452) As a follow up to \#22445, this extracts the queueing logic that is shared between ‘dispatchSetState‘ and ‘ dispatchReducerAction‘ into separate functions . It likely doesn’t save any bytes since these will get inlined, anyway, but it does make the flow a bit easier to follow.	There is repeated queueing logic in ’ dispatchSetState’ and ’dispatchReducerAction’, making the code difficult to follow.
fix(ivy): classes should not mess up matching for bound dir attributes (\#30002) Previously, we had a bug where directive matching could fail if the directive attribute was bound and followed a certain number of classes. This is because in the matching logic, we were treating classes like normal attributes. We should instead be skipping classes in the attribute matching logic. Otherwise classes will match for directives with attribute selectors, and as we are iterating through them in twos (when they are stored as name-only, not in name-value pairs), it may throw off directive matching for any bound attributes that come after. This commit changes the directive matching logic to skip classes altogether. PR Close \#30002	Directive matching fails when the directive attribute is bound and follows a certain number of classes, as classes are mistakenly treated as normal attributes in the matching logic.
KAFKA-13396: Allow create topic without partition/ replicaFactor (\#11429) {[}KIP-464{]}(https://cwiki.apache.org/confluence/display/KAFKA/KIP-464\%3A+Defaults+for+AdminClient\%23createTopic) (PR: https://github.com/apache/kafka/pull/6728) made it possible to create topics without passing partition count and/or replica factor when using the admin client. We incorrectly disallowed this via https://github.com/apache/kafka/pull/10457 while trying to ensure validation was consistent between ZK and the admin client (in this case the inconsistency was intentional). Fix this regression and add tests for the command lines in quick start (i.e. create topic and describe topic) to make sure it won’t be broken in the future. Reviewers: Lee Dongjin \ textless{}dongjin@apache.org\textgreater{}, Ismael Juma \textless{}ismael@juma.me.uk\ textgreater{}	Inconsistency between validation in ZK and the admin client after applying KIP-464, resulting in inability to create topics without passing partition count and/or replica factor.

Table 6: Examples of transformed queries for one sample commit from facebook_react, angular_angular and apache_kafka respectively.