International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 10 Issue: 10 | Oct - 2026 SJIF Rating: 8.659 ISSN: 2582-3930

Hybrid Retrieval and AI-Based Frameworks for Software Repository Understanding

Mrunal Mohite[1]; Piyush Shelke[2]; Maharudra Kadam[3]; Prof. B. S. Kulkarni[4]
[4]Professor
[1,2,3,4]Department of Computer Engineering, Pune Institute of Computer Technology, Pune, Maharashtra, India

Abstract: Developers working in software repositories often have a hard time finding the right code and understanding how files are connected. Keyword search can miss the meaning of a query and purely semantic search might ignore exact names or identifiers. This review explores retrieval and retrieval-augmented generation or RAG for better understanding of code repositories. It covers semantic search methods, how code is represented and split into chunks context at the repository level and answers that are supported by evidence. The studies show that using retrieval methods adding reranking and grounding answers in retrieved sources leads to more dependable results. Still there are challenges. Keeping indexes up to date as code changes is difficult. Processing costs can be high. Handling dependencies across files remains tough. And support, for programming languages that're less common is still weak.

Keywords: Hybrid Retrieval, Retrieval-Augmented Generation, Code Search, Abstract Syntax Trees, Reciprocal Rank Fusion, Software Repositories.

I. INTRODUCTION

Large software repositories contain millions of lines of code that are spread across thousands of files. They also include dependencies build scripts, schemas and documentation. Figuring out how all of these parts fit together is rarely a task. Good software design separates functionality into modules. This separation helps teams collaborate and keep code under control. Separating code into modules also means that understanding one piece of code often requires seeing context in other files. Searching for code is still not always satisfying. This is partly because the words people use when asking a question often do not match the words that appear in the code. Tools that learn from code collections try to increase programmer productivity and improve software quality. A better understanding of a repository can help developers work efficiently. A better understanding can also help developers keep software running.

For a time developers have used text search tools like grep, regular expressions and the built-in search features in their IDEs to look through code. These tools are good at finding words, tokens or symbol declarations. They fall short when a developer asks a question in plain language like where user authentication is checked or how temporary database connection failures are handled. The problem comes from a mismatch. Developers describe what they want in language but the code uses abbreviations, project-specific names and framework conventions. This means there is often overlap between the words used in the search and the actual code. Code written by a person also follows naming habits and programming styles. That makes matching based on words unreliable. That is why research in code search has shifted toward learning from pairs of code and natural-language descriptions. The goal is to match a query to a function based on meaning, not, on spelling.

Two further motivations are project-specific. The first motivation is knowledge retention. When the original authors move on the reasoning behind a subsystems design, legacy API workarounds and hidden constraints can be easily lost. Some of that reasoning lives in README files and configuration files [10]. By indexing commit messages, pull-request reviews and issue discussions next to the source tree we can retrieve intent. The second motivation is that the code at hand is often incomplete. A file being edited may not. A full build is rarely available across several repositories. Compiler-based analysis needs code that can build while retrieval can still match code by meaning. Parsers can handle snippets [8]. Retrieval is therefore well suited, for early-stage completion and exploratory search.

This gap is addressed by code search. It maps queries and code into a shared embedding space so they can be matched by meaning. Examples include the bimodal pre-training in CodeRetriever [7] and the dependency-aware encoding in CRaDLe [12]. Dense retrieval however fails in the direction. A fixed-size vector cannot hold every detail of a token sequence. So an exact variable name, a flag or a one-character

difference is easily lost [1] [11]. To cover these weaknesses hybrid retrieval has become a common approach. It combines dense rankings. Then it passes the result to a language model, in a retrieval-augmented generation (RAG) pipeline [3] [4] [11].

Applying RAG to code repositories brings challenges that ordinary text retrieval does not have. First code has a structure so simple line or character windows often break up functions and class definitions and cutting code in random places can lose important information [9]. Second a repository is connected through dependencies so looking at files one, by one hides how classes inherit from each other and how functions call each other [8] [12]. Third generative models can create method signatures that do not exist and make up details in private repositories where the naming rules are unknown [8] [4]. This review looks at how research deals with these issues and is organized around four research questions (RQs).

RQ1 (Retrieval synergy): How do sparse lexical and dense semantic retrievers complement each other on codebases, and which fusion mechanism gives the best ranking? **RQ2 (Syntactic chunking):** How does chunking along AST boundaries compare with line-based windows in keeping code semantically coherent? **RQ3 (Repository-scale context):** Which mechanisms handle cross-file dependencies and call chains? **RQ4 (Attribution and grounding):** How can generated explanations be constrained and evaluated so that they remain verifiable and traceable to specific sources?

The main contributions of this paper are as follows. First we present a four-pillar taxonomy that organizes hybrid retrieval and AI-based repository understanding. Second we gather the ranking fusion, chunking and evaluation equations in one place. Third we compare systems and collect the quantitative results their authors report. Finally we examine infrastructure, -agent and evidence-based software engineering (EBSE) perspectives [5] [6] along, with the open research gaps that still exist. The rest of the paper follows the structure. Section II reviews the literature Section III explains the review methodology and formal foundations Section IV presents the results Section V discusses the implications and gaps and Section VI concludes.

II. LITERATURE REVIEW

➢ *Sparse Lexical Retrieval*

Lexical retrieval is based on the relevance framework. The well-known ranking function in this area is Okapi BM25 [11]. When a query Q = {q1, ... qn} is matched against a document D the score is built from three parts. First how rare each query term is across the collection. Second a saturating function that looks at how the term appears in the document. Third a normalization factor that adjusts for the length of the document. These are shown in equations (1)-(3). The second and third parts are key, to BM25's success. Term-frequency saturation and document length normalization are often said to be the reasons why BM25 works well [11]. That's why BM25 is still used as a baseline. It also holds up well against models that are trained using machine learning [1] [3].

Typical settings are k_1 between 1.2 and 2.0 and b = 0.75.

$$\mathrm{BM25}(D,Q) = \sum_{i=1}^{n} \mathrm{IDF}(q_i) \cdot \tau(q_i, D) \quad (1)$$

$$\tau(q_i, D) = \frac{f(q_i,D)(k_1+1)}{f(q_i,D)+k_1\left(1-b+b\cdot\frac{|D|}{avgdl}\right)} \quad (2)$$

$$\mathrm{IDF}(q_i) = \ln\left(\frac{N-n(q_i)+0.5}{n(q_i)+0.5}+1\right) \quad (3)$$

Here f(qi, D) is the count of the term in the document |D| is the document length, in tokens avgdl is the average document length N is the number of indexed files or chunks and n(qi) is how many of those contain the term. Sparse engines build indexes that map each term to a list of the places it appears which keeps queries fast even over large collections [1] [11].

Tokenizing source code has its challenges. Identifiers are written in camelCase or snake_case so a tokenizer that splits them into sub-words while keeping the token allows BM25 to match queries with exact symbol names, method calls or configuration keys very accurately. CodeSearchNets baselines deal with this by splitting identifiers into sub-tokens and its keyword-based ElasticSearch baseline remained competitive partly because keyword search works with rare terms that appear often in code [1]. The same feature is also a drawback. BM25 relies on word overlap so a query like "abort pending network request" cannot find a method written as client.cancel_call(). Rare identifiers also get high inverse document frequency weights while common verbs such, as get, set and parse are ignored.

➢ *Dense Semantic Retrieval*

Dense retrieval turns queries and passages into vectors and ranks them based on how close they are in that vector space. Since both the query and the code are turned into one vector each we can precompute all the vectors for the whole collection. Then during search we just compare them quickly. This makes indexing. Searching much more efficient [1]. CodeRetriever uses encoders and measures the score of a pair using cosine similarity [7]. The enterprise system in [11] also uses sentence embeddings with a FAISS index, for the reason. To speed things up.

$$\mathrm{Cos}(e_q, e_d) = \frac{e_q \cdot e_d}{\|e_q\|\|e_d\|} \quad (4)$$

DPR uses two transformer encoders. One is for queries and the other is for passages. It scores a query-passage pair by taking the product of their encoded representations. During training it combines negatives that're already in the batch, with BM25 hard negatives. These hard negatives are passages that share words with the query but do not have the answer. This approach beat BM25 on open-domain question answering tasks. The same research shows the time it takes to prepare the system. Encoding 21 million passages took 8.8 hours using eight GPUs. After that building the nearest-neighbor index needed another 8.5 hours.

A bi-encoder looks at the query. The passage on its own. Because of that it might miss the connections between words in the query and parts of the code. This is an issue and papers [7] and [11] point it out. Cross-encoders fix this by reading the query and the candidate at the time. That lets them interact deeply which helps them be more accurate. Paper [11] talks, about this benefit.. There's a price. The more candidates you check the more time and resources it takes. So in real-world systems people often use a two-step method. First a fast cheap step picks out a group of likely candidates. Then a cross-encoder steps in to rank those top more carefully. This approach is common. Papers [9] and [11] support it.

Hybrid retrieval and reranking influence one another. In candidate snippets from three retrieval paths – sparse, dense and dataflow-guided – are collected together and then reranked because a single retriever might not order results the way the code model likes [9]. The enterprise pipeline in [11] does the same: it runs sparse retrieval at the same time combines the outcomes and reranks the merged list. In practice hybrid retrieval and reranking runs both retrievers over the pieces of text merges their outputs by rank or by a normalised score and adjusts the fusion weights using held-out queries. The weights, in [11] were set at 0.6 and 0.4. The best values can change from one corpus to another. Therefore hybrid retrieval and reranking should be checked for each repository. Not taken from another benchmark.

➢ *Code-Specific Models and Benchmarks*

I noticed that CodeSearchNet [1] gave semantic code search a yardstick. Its corpus holds about 6.4 million open-source functions in Go, Java, JavaScript, PHP, Python and Ruby and roughly 2.3 million of them carry a docstring that works as a plain-language description. When that docstring is used as the query a self-attention encoder ranked best with a mean reciprocal rank (MRR) of 0.7011. On the 99 expert-judged Challenge queries however a simple bag-of-words model came out ahead. An ElasticSearch keyword baseline stayed competitive. The authors link this to the gap between docstrings and real developer queries [1]. Wan et al. [2] Likewise found that bag-of-words came close, to encoders at far lower cost, which is one reason hybrid designs make sense.

CodeBERT is a 12-layer bimodal transformer. It is pre-trained on paired natural-language documentation and code. It uses masked language modeling. Replaced token detection as its training objectives. On the CodeSearchNet dataset CodeBERT achieved a MRR of 0.7603. This is higher than RoBERTa, which scored 0.6972. Probing experiments showed that CodeBERT learns -modal alignments between code descriptions and program logic. Graph CodeBERT extends this idea by incorporating data-flow structure into the attention mechanism. A broader survey and toolkit outline how deep learning, for code intelligence is structured across tasks and representations. Code can be represented in ways, including sub-token sequences, abstract syntax trees (ASTs) control-flow graphs and data-flow graphs. Structure-aware models benefit from the three representations using the structural information to improve performance.

I have found that code can be written in ways and each way determines what a model can see. A string of sub-tokens keeps naming habits. It loses structure. An abstract syntax tree (AST) shows nesting, a control-flow graph indicates which statements may follow which and a data-flow graph connects each use back to the assignment that may have set its value [2]. Wan et al. [2] Point out that most language models still learn from token sequences so giving them this structure is still an open problem. CodeRetriever [7] follows one path, on the model side by starting from GraphCodeBERT which introduces data flow into code pre-training. Structure can also be added at the index. DraCo [9] builds a graph of the entire repository and CodeRAG [10] splits code along AST boundaries instead of using fixed lengths. These two paths complement each other because even a structure-aware encoder only sees the chunk it receives and a cut chunk wastes most of what the encoder can do.

➢ *Repository-Level Completion and Chunking*

Most completion tools usually work inside a single file and do not see the context that spreads across the rest of a project. Even though completion tools can be useful they often miss context. I have seen that RepoCoder was an answer to this problem and later studies use RepoCoder as a main baseline [9] [10]. RepoCoder cuts the repository into sliding windows. Retrieves snippets that look similar to the unfinished code. RepoCoder then retrieves a time adding the models own first draft to the query so the search can reach cross-file context that the unfinished

code alone could not [9]. The extra round does help. On CrossCodeEval with StarCoder-15.5B the iterative version raised match from 26.27% for a single retrieval to 29.12% and the zero-shot figure was only 8.71% [9]. The price is speed because the second retrieval has to wait for a draft. DraCo measured four seconds to build a prompt even with the small CodeGen-350M model, which is too slow, for real-time completion [9].

The RepoCoder loop can be described in five steps. First a similarity retriever pulls candidate snippets C0 for the prefix P. Next the code model writes a hypothesis Y0 based on P and C0. The system then builds an expanded query by attaching Y0 to P, which brings in the API calls and variable names the model expects to see. The retriever runs again with this query to gather better cross-file context C1 and the model finally produces the completion using P and C1. The key insight is that the models own draft even if imperfect sits closer in vocabulary to the target code, than the prefix so it works as a stronger query.

Sliding windows still cut through syntax. Strich et al. [11] For example split files into 1,500-character chunks with a 200-character overlap and attach metadata such as file name, line numbers and imports. Newer systems cut along structure instead. CodeRAG [10] parses each file into an AST. Keeps whole functions global variables, class variables and class methods as its knowledge-base units arguing that arbitrary splitting loses details, such as when a class is divided. DraCo [9] parses repositories into modules, classes, functions and variables links them through dependency relations and writes the retrieved entities back out as code. Neither paper tests fixed windows, against AST chunks directly. The benefit remains plausible but unmeasured.

An AST-based chunker can be created using ideas that have already been used in the systems that were reviewed. The parser starts by converting each file into an AST. Tree-sitter is an option because it can also work with code that is not complete [9]. Each top-level function, class method, variable and class variable then becomes one separate unit just like in CodeRAG [10]. Each unit keeps information about its location, such as the file name, the module, the starting and ending lines and the imports, which Strich et al. [11] Store with each chunk. DraCo [9] also makes sure not to repeat a class member if the parent classs already in the context. It also adds a file-path comment before each module so the prompt shows the directory structure. An enclosing-scope header, on every chunk uses the idea. Splitting a long method at loops and conditionals or combining small nearby parts is our own idea and has not been tested in [9] [10] or [11].

Cross-file relationships can also be shown using static call graphs. These graphs have functions as points and function calls as arrows. When a function is found the system can look at who called it and who it called for a number of steps. This helps get information about the codebase. The Jaccard coefficient can then check the sets of execution paths between parts. This helps find components that're similar, in structure even if they don't share many words.

➢ *Query Understanding and Pre-Retrieval Optimization*

Before retrieval I see that RAG pipelines often rewrite or expand the query send it to an index or build a hypothetical answer and use its embedding for the search. For repositories I find three adaptations that seem plausible. Identifier extraction pulls out tokens that look like symbols, such as CamelCase names, dotted paths and error constants and keeps them apart from words so the lexical and dense branches can weight each group differently. Query expansion with synonyms of programming verbs narrows the vocabulary gap; for instance linking cancel with abort helps. Editor context like the file its imports and recent edits can be added to the query following the same intuition, behind RepoCoders use of the unfinished prefix. After retrieval context compression and de-duplication trim the prompt, which matters because of bias. These steps cost little compared with generation and steps are often the place to look when a hybrid pipeline underperforms.

➢ *RAG Taxonomy and Attributed Generation*

Retrieval-augmented generation (RAG) brings together a language model and an outside document index. This allows responses to use information that is found of only using knowledge that is already inside the model [3] [4]. RAG methods are usually divided into three types. Naive RAG uses a retrieve-then-generate method. However this method can be limited by information that's not relevant or not complete. Advanced RAG makes this better by using techniques such, as changing the query before looking for information and then reordering the results or making the context shorter. Modular RAG takes this further by using parts. These parts include ways to get information that change depending on the situation. They also include searching than once and improving the results based on feedback. This helps to solve complicated information needs.

Users can better trust generated answers when the information is supported by relevant, verifiable evidence. Retrieval-augmented generation (RAG)

helps connect a model's responses to retrieved information, while research on evidence-based software engineering highlights the importance of grounding AI-generated outputs in reliable evidence [3]–[5]. In software repositories, this means connecting answers to relevant code and project documentation so developers can check the supporting information themselves [11]. However, retrieving relevant material is only part of the solution; the system must also ensure that the evidence genuinely supports the generated answer.

➢ *Evidence-Based Software Engineering*

EBSE bases. Architecture choices on secondary studies but systematic literature reviews (SLRs) take a long time and are seldom read by people who work in practice. Guidelines for reviews in software engineering explain the protocol that most reviews use. A new vision paper talks about how AI could help with each step of an SLR starting with creating search strings based on PICO to choosing studies and checking their quality. The paper also points out problems with access to libraries unpredictable results from AI and the importance of having humans check the work. A registered report suggests a study about one-page summaries made by language models using a vector store and a crossover experiment setup comparing how well the summaries keep the original content compared to summaries written by people. A long-term study of teaching EBSE shows that when developers get training in appraisal and synthesis their way of looking for information gets better. A known problem, with summaries made by AI is that findings that depend on conditions are said as if they are always true so the process of making summaries needs clear places where sources are cited and needs to have expert checks.

III. METHODOLOGY

➢ *Review Protocol and Search Strategy*

This review follows the systematic literature review principles that Kitchenham and Charters have set out. It has been adapted to study generative AI in software engineering. The systematic literature review looked at papers from IEEE Xplore, the ACM Digital Library, the ACL Anthology and arXiv. The systematic literature review covered three areas. First software repository analysis, which includes code search and repository-level completion. Second retrieval techniques, which include dense retrieval, BM25, dual encoders and reciprocal rank fusion. Third AI-based methods, which include retrieval-augmented generation, large language models, code embeddings, AST-based chunking and source attribution. However the systematic literature review does not state the search dates the number of records that were found and excluded at each screening stage or a formal quality-assessment process. These gaps, in the literature review reduce the reproducibility of the review and should be filled in whenever the needed information can be obtained.

➢ *Inclusion and Exclusion Criteria*

A study was included if it (i) was a peer-reviewed or foundational pre-print publication that introduced an algorithm, dataset, or evaluation framework for source-code retrieval or repository understanding; (ii) investigated hybrid combinations of lexical and dense retrieval; (iii) addressed structural chunking, AST representations, or cross-file context; or (iv) evaluated attribution fidelity or citation verifiability. A study was excluded if it was limited to single-line in-file completion without repository context, if it covered open-domain question answering with no applicability to programming languages, or if it described a proprietary closed-source tool without a public benchmark or empirical baseline.

➢ *Taxonomy*

The selected studies were grouped into four pillars, which are displayed in Figure 1: sparse code search (Okapi BM25 and inverted-index optimization) dense semantic code encoders (SBERT, DPR, CodeBERT, CodeSearchNet) repository-level structuring and chunking (RepoCoder, CAST, AST and call-graph analysis) and grounded AI synthesis with evidence attribution (ALCE, Modular RAG, evidence briefings). Infrastructure topics such as vector indexing, quantization and multi-agent orchestration span, across all pillars and are discussed in Section V.

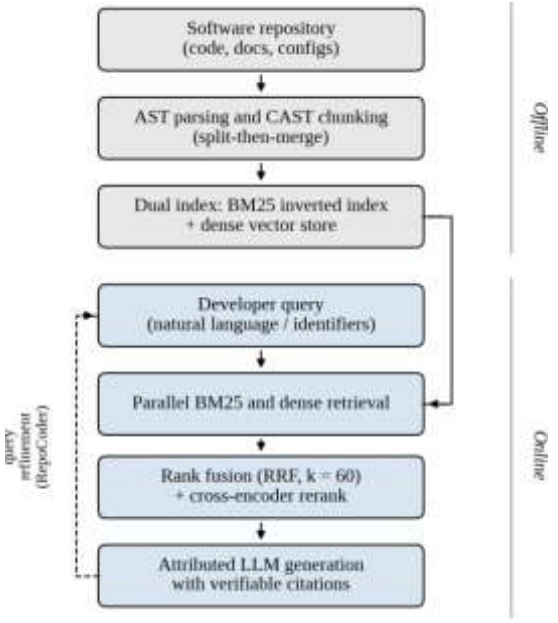


Fig 1 Architecture Of A Hybrid Retrieval And Ai-Based Repository Understanding Pipeline.

➢ *Data Extraction and Synthesis*

For each study I extracted the task, the dataset, the retrieval and fusion method the chunking granularity, the generator, the evaluation metrics and the stated limitations. The findings were then assembled into a narrative organized by pillar and, by research question. I only put results into tables when the original authors made them explicit and I ensured that every table entry was clearly linked to its source.

➢ *Formal Foundations*

Because BM25 scores and neural similarities sit on numeric scales a linear hybrid first does min-max normalization, to each signal. Then the linear hybrid mixes the normalized scores using convex weights. The signals that the linear hybrid can use may be BM25 scores, similarity, an AST-based signal and a freshness signal.

$$x' = \frac{x - min(X)}{max(X) - min(X)} \quad (5)$$

$$S_{\text{hyb}} = \sum_{j=1}^{m} w_j S_j' \quad (6)$$

Reciprocal Rank Fusion, also known as RRF does not need calibration. RRF works by combining ranks of combining raw scores. Suppose we have a set of rankings called R and let r(d) denote the 1-based rank of a document d within a ranking r. The combined score given by Reciprocal Rank Fusion is expressed in equation (7). In that equation the smoothing constant, denoted k is usually chosen to be 60. If a document does not appear in a ranking that ranking does not add any value to the document's overall score.

$$\text{RRF}(d) = \sum_{r \in R} \frac{1}{k + r(d)} \quad (7)$$

For chunking the CAST merge rule takes sibling AST nodes that have the same parent and puts them into one chunk. This happens long as the total length of these nodes does not go over the limit called Cmax as shown in equation (8). When it comes to checking how good the retrieval is we use Precision@k, Recall@k normalized discounted gain and MRR (9) on a set of queries called Q. For each query i ranki is the position of the first result that is relevant. For generation we check with match, edit similarity and execution-based Pass@k. For attribution we look at ALCE citation recall and precision.

$$W_j = \bigcup_{i=a}^{b} n_i \; s.t. \sum_{i=a}^{b} l(n_i) \le C_{\max} \quad (8)$$

$$\text{MRR} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\text{rank}_i} \quad (9)$$

Structural similarity between two modules, represented by their sets of execution paths or imports, can be measured with the Jaccard coefficient (10).

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \quad (10)$$

This is a narrative systematic review and not a meta-analysis. The figures we report are those published by the original authors under different datasets, models, and settings, and they are therefore not directly comparable across studies.

IV. RESULTS

➢ *Comparative Analysis of Representative Systems*

Table 1 shows the systems that anchor this review along granularity, retrieval or fusion strategy and principal limitation. I observe that three patterns stand out. I find that every retrieval-only method has a blind spot: lexical methods miss paraphrase dense methods miss rare tokens and neither sees structure. I notice that methods that add structure either through ASTs or through iterative query refinement improve context quality but pay, in parsing cost or inference latency. Finally I see that attribution methods improve verifiability but depend on the quality of the retriever and of the NLI judge.

Table 1 Comparison Of Representative Systems

System	Granularity	Retrieval / fusion	Principal limitation
BM25	Document or chunk	Sparse inverted index	Vocabulary mismatch; no semantics
SBERT	Sentence	Siamese encoder, cosine	Unaware of code syntax
DPR	Passage	Dual encoder, inner product	Costly indexing; weak on rare tokens
CodeBERT	Token / function	Bimodal transformer	Limited context window; no hierarchy
RRF	Ranked list	Parameter-free rank fusion	Ignores score margins
RepoCoder	Sliding window	Iterative query refinement	Latency; unstable if early draft is wrong
CAST	AST node	Split-then-merge chunks	Parsing cost; grammar per language
ALCE	Passage	Dense or BM25 + citations	NLI misses partial support
Modular RAG	Chunk	Hybrid, rerank, routing	Orchestration latency; tuning
Evidence briefings	Paper section	Vector store, in-context	Small evaluations; prompt-sensitive

➢ *Consolidated Quantitative Findings*

Table 2 summarizes the quantitative results reported in the selected studies, while Figs. 2 and 3 present the findings visually. On code search benchmarks, CodeBERT achieved an average MRR of 0.7603, compared with 0.6972 for RoBERTa, indicating improved performance from pre-training on both code and natural language [7]. At the

repository level, iterative retrieval and generation methods aim to improve code completion by incorporating relevant information from across a project [9], [10]. Research on evidence-grounded generation also highlights the importance of connecting answers to supporting sources, making verification easier [3], [4]. However, retrieval quality and computational cost remain important considerations, particularly when indexing large collections of code and documentation.

Table 2 Results Reported in the Original Studies

Task / metric	Baseline	Method	Ref.
CodeSearchNet, avg. MRR	RoBERTa 0.6972	CodeBERT 0.7603	[5]
CodeSearchNet, avg. MRR	Self-attention 0.7011	CodeBERT 0.7603	[2], [5]
RepoEval, unit-test pass rate	23.32% (in-file)	42.63% (RepoCoder)	[9]
ALCE ASQA, citation recall	26.7% (closed-book)	73.6% (in-context)	[10]
Sentence-pair comparison time	About 65 h (BERT)	About 5 s (SBERT)	[3]
DPR index build (21M passages)	-	8.8 h encode + 8.5 h index	[4]

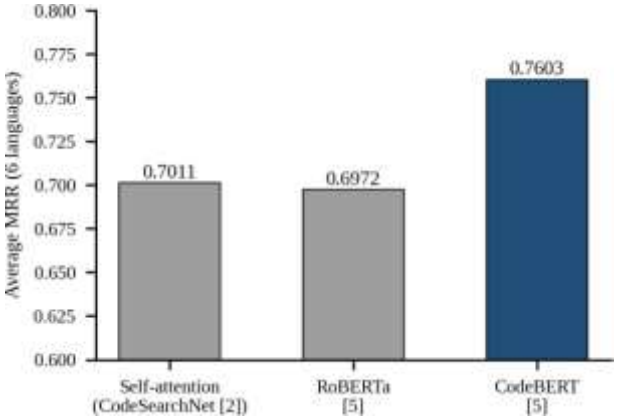


Fig 2 Average MRR On Codesearchnet (Axis Truncated At 0.60).

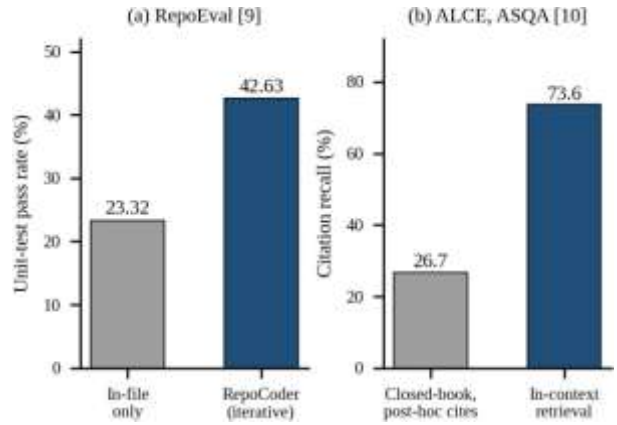


Fig 3 Reported Effect Of Iterative Retrieval And In-Context Citation.

These numbers need to be looked at. They are from tests so the actual numbers can't be compared from one line to another. Some of the improvements are because of tools used for creating and finding information. For CAST the original research shows results when getting and making information with different tools, on RepoEval and SWE-bench. We don't show one number because how much it improves depends on the tools used.

➢ *Answers to the Research Questions*

RQ1. Sparse and dense retrievers fail on complementary queries. BM25 tends to work best on identifiers, error strings, and exact API parameters, while dense encoders tend to work best on paraphrased intent. Rank-based fusion with RRF is robust because it needs no score calibration and helps prevent a single noisy retriever from dominating. Linear fusion can be more expressive but is sensitive to score distributions and to drift across repositories. A cross-encoder reranker over the fused candidates adds precision at the price of latency.

RQ2.AST-aware chunking keeps functions, classes, and signatures intact, whereas line-based windows cut right through them. The split-then-merge procedure handles the opposite problem of AST fragmentation, where single statements turn into micro-chunks that inflate the index and water down the context. The trade-off is the extra parsing overhead and the reliance on language-specific grammars.

RQ3. Cross-file dependencies are tackled through three families of mechanisms: iterative query refinement using model drafts, explicit call-graph expansion over static analysis output, and hierarchical or multi-agent decomposition (Section V). None of them comes without a cost. Iteration multiplies the inference cost, graph expansion can flood the prompt, and long prompts are prone to positional bias.

RQ4. Verifiability calls for retrieval inside the generation context, explicit numeric citations, and automatic checking with entailment models. ALCE shows that adding citations after the fact to closed-book answers is unreliable . For repositories, citations need to resolve to file paths, declarations, and line ranges, and NLI verification should sit inside the response pipeline rather than being left to an offline audit.

➢ *Cost and Latency Profile*

Quality gains come with cost that varies by part. Word search is the part because its small index is fast to change. Dense search adds an embedding step at index time and an approximate nearest-neighbor search at query time and its index becomes much

larger than an one. Cross-encoder reranking is used on the top K combined candidates but cost grows straight with K and each candidate requires a full pass over the question and the chunk. Iterative generation multiplies the cost of all: a two-step RepoCoder run needs at least two search passes and two model calls while a single pass needs only one. A rough end-to-end latency model is therefore the total of the search. Merge time K times the cost, per candidate reranking and the number of steps times the generation time. Because this total cost must stay within limits designers should use their latency budget on the parts the data says are most valuable namely hybrid candidate generation syntax-aware chunks and in-context citation.

➢ *Failure Modes of Each Retrieval Style*

Looking at where each approach doesn't work helps explain why combining them works. Lexical retrieval stops working when the question doesn't use any of the words as the target, when the tokenizer splits names in different ways or when a common word like get or set matches many parts and hides the right answer. Dense retrieval has problems in the way. Two functions that are the same except for a small change like a negation or a version number end up looking very similar in the embedding space so the system can't tell them apart. Words that aren't in the vocabulary get split into parts that don't make sense. A system that uses both methods takes advantage of the parts of each but doesn't automatically fix the problems they both have. A part that both systems miss will still be missed which is why making sure the parts are good and rewriting the questions is just as important as how the two systems are combined. Rerankers add another problem because a cross-encoder trained on one language or style might give scores to parts from another. Chunking that pays attention to syntax reduces errors that start with boundaries and checking who wrote the part catches errors that start with unsupported results but neither fixes when there are no results at all. Systems in production should therefore record the questions that don't get any part above a confidence level and send them to a method, like simple keyword search or a clear message that no information was found.

V. DISCUSSION

➢ *Lexical Precision versus Semantic Generalization*

The main design problem is that code is a language where names have exact meaning while developers usually ask in informal language. Retrieval is key. A hybrid pipeline helps to reduce this problem. Only if a fixed fusion rule is used. Query-aware fusion, which gives BM25 weight for queries that have many identifiers and gives dense retrieval more weight for conceptual queries seems like a natural next step although we did not find a mature and widely validated method of this type in the literature we examined. RRF also discards score margins so it cannot show how sure a retriever is, about its top hit. Table 3 shows how the three retrieval styles compare qualitatively.

Table 3 Qualitative Comparison Of Retrieval Styles

Criterion	Lexical	Dense	Hybrid + rerank
Exact identifiers	Strong	Weak	Strong
Conceptual queries	Weak	Strong	Strong
Query latency	Lowest	Low	Highest
Index memory	Low	High	High
Score calibration	Not needed	Not needed	Needed (or use RRF)
Structure awareness	None	Sequence only	Via AST / graph

➢ *Syntactic Integrity versus Context Budget*

Plain-language documents have boundaries. Splitting them by characters or by sentences is usually enough. Code is built from scopes and inheritance. Because of that a random cut can separate a signature from its docstring or a variable from its definition. Syntax-aware chunking can solve the problem.. It needs a parser, for each language in a polyglot repository. Also configuration formats and multi-language bindings usually do not have a grammar. A practical workaround is to put the enclosing namespace and class signature at the front of each chunk. This way inner blocks still make sense when they are placed on their own.

➢ *Cross-File Context and Latency*

Iterative generation and graph expansion both expand the context. RepoCoder points out that later iterations can perform worse if the early drafts have mistakes. Long prompts can also make it hard to find the information. Finding graph-aware context subgraphs that follow call chains and type definitions within a limited number of tokens is still a key open challenge. In IDEs there are timing limits—about 200 milliseconds for search and, around one second for inline completion [10]. These limits make it difficult to use much reranking or many iterations.

➢ *Vector Infrastructure and Quantization*

At enterprise scale, memory is often the main constraint. A 768-dimensional float32 embedding takes up 3,072 bytes, so ten million code chunks need about 30.7 GB of raw vector storage. Hierarchical Navigable Small World (HNSW) graphs offer fast approximate search with high recall, though they cost extra memory for the graph. Product quantization (PQ) slices a D-dimensional vector into M sub-

vectors and swaps each one for the index of its nearest centroid out of 256, so a vector occupies just M bytes. With M = 96, a 768-dimensional float32 vector drops from 3,072 to 96 bytes, a 32-fold reduction. Asymmetric distance computation compares the uncompressed query against the stored centroids through lookup tables, so nothing has to be decompressed . Scalar quantization to 8 bits trims storage by 75% with little loss of recall in typical settings, and IVF-PQ and HNSW can be used together. The best configuration depends on the target and the available RAM, so each deployment should be benchmarked on its own corpus.

➢ *Multi-Agent Orchestration and Protocols*

As tasks grow to include refactoring, system-wide bug fixing, and security auditing, single prompts hit their limits, and multi-agent designs split the work up. The reviewed material discusses four topologies [6]. In a centralized topology, a supervisor breaks a request down and hands the pieces to specialist agents for parsing, hybrid retrieval, code generation, and test execution. In a decentralized topology, agents publish and consume events over a message bus. In a hierarchical topology, team-level supervisors oversee subsystems under a top-level coordinator. In a blackboard topology, specialists read from and write to a shared memory of call graphs, test results, and open defects. A critic or reviewer agent can check candidates against static analysis, compiler output, and unit tests, which closes the loop that purely passive RAG is missing.

Interoperability is being standardized.The Model Context Protocol or MCP sets up a client-server boundary that lets orchestrators reach prompts, resources and tools such as AST parsers, static analysers and execution sandboxes.The Agent2Agent or A2A protocol defines JSON-RPC style messaging, between agents.These topologies and protocols are still young and we found controlled evidence comparing them so we treat them as promising designs rather than established best practice.

➢ *Evidence Briefings, Security, and Licensing*

In EBSE retrieval-augmented pipelines can handle search-string generation, abstract screening and writing evidence briefings. These tools can save time and effort.. There are risks. Sometimes they make up numbers during data extraction. They might also be too sure of themselves when putting things together leaving out caveats.. They can wrongly exclude studies during screening. That's why human review is still needed. We must keep using citations to back up claims.For generated code repository-grounded pipelines can check suggested code, against indexed sources. They can spot license issues like copyleft code or known security weaknesses before a commit. This kind of check is helpful and growing in value. It doesn't take the place of real legal or security review. Humans still need to step in for those parts.

Table 4 maps the SLR stages discussed in the EBSE vision paper to the generative AI support that each stage can receive and to the principal risk that reviewers should watch for. The pattern is consistent: the closer a stage is to producing a number or a conclusion, the more important it is to keep a human in the loop and to keep every statement traceable to a passage in the source study.

Table 4 Generative AI Support Across SLR Stages

SLR stage	Generative AI support	Principal risk
Search strategy	PICO-based query expansion	Overly broad search
Title/abstract screening	Zero-shot classification	False exclusion
Quality appraisal	Prompted rubric evaluation	Overrating weak studies
Data extraction	Form-filling with RAG	Hallucinated numbers
Evidence synthesis	Multi-document summarization	Overstated certainty

➢ *Evaluation Practice*

Evaluation methods can strongly affect how research findings are interpreted. Surface-level metrics, such as match and edit similarity measure how closely generated code resembles a reference. However evaluation methods do not necessarily show whether the code works correctly. Execution-based measures, such as Pass@k provide a practical assessment by checking whether generated solutions pass unit tests [9] [10]. For retrieval Mean Reciprocal Rank (MRR) measures how highly the first relevant result appears, while Recall@k indicates whether useful evidence is present among the retrieved results. Citation precision and recall are also important because a fluent answer may still rely on insufficiently supportive sources [3] [4]. To make evaluations more meaningful studies should compare system components through ablation tests, report latency and memory use alongside accuracy and test on repositories not used during model training whenever possible. Including developer questions, from the target codebase can further reveal practical limitations that standard benchmarks may not capture.

➢ *Research Gaps and Limitations*

Table 5 lists the gaps we see as most important. Two limitations of this review should also be noted. First, the evidence base leans heavily on Python benchmarks and a small number of generator models, so the conclusions may not carry over to other languages or to newer models. Second, the reported results come from heterogeneous setups, so our

consolidation shows the direction and size of the effects but doesn't rank systems.

Table 5 Open Research Gaps

Gap	Why it matters	Direction
Incremental indexing	Full re-indexing is prohibitive	Delta-indexing of changed AST subtrees
Adaptive fusion	Fixed rules ignore query type	Query-aware weighting
Graph-aware context	Prompts overflow, evidence lost	Compact call-graph subgraphs
Verifiable citations	Partial support is missed	Line-level NLI in pipeline
Low-resource languages	Benchmarks favor Python	Language-aware parsers, data
Local deployment	Privacy and latency	Quantized retrievers, small LMs

➢ *Reference Design for Practitioners*

The reviewed evidence points to the following default architecture for a repository assistant. It is a summary of the cited studies. Was not tested experimentally in this review. Offline, the system parses each file with a language grammar chunks it with a split-then-merge method under a character limit adds scope headers and stores the file path, line range, language and commit together with each chunk so that citations can be shown exactly. It builds an index over identifier-split tokens and a dense vector index over the same chunks. Online the system runs both retrievers in parallel to get a candidate pool of a dozen to a hundred chunks each fuses the lists with RRF and re-ranks a smaller top set with a cross-encoder when the latency budget allows. It expands with call-graph neighbors for the few best hits to protect the prompt budget. It generates text with citations checks each claim against its cited chunk and removes any statement that lacks support. It keeps iterative query refinement for completion tasks where the first retrieval passs weak. It quantizes vectors when memory is tight re-indexes the changed files, on each commit and tracks MRR and execution-based pass rates on a small gold set of real developer questions drawn from the target repository.

➢ *Threats to Validity*

Threats exist for this review. I looked at four sources so threats are high that I missed other work in different places or languages. Many benchmark papers show good results, which creates a threat of bias. Threats, from benchmark tasks are that they may not match what real developers ask because many CodeSearchNet queries come automatically from comments. Another threat is that pre-trained models may have already seen the benchmark repositories during training, which can make the scores look better than they truly are.

Finally, closed-source generators change over time, which limits how reproducible some of the results are.

➢ *Responsible Use*

Repository assistants often handle code, so privacy and compliance become key points when they are put into use. Repository assistants must keep the source code safe. Sending source code to third-party cloud services can raise worries about protecting data and keeping property safe. Therefore putting the system on a machine and using quantized retrieval models and smaller language models can be a good alternative. Access controls must be applied during retrieval so that users cannot see files they are not allowed to see. Even though grounding generated responses, in retrieved evidence can make answers more reliable it does not remove mistakes entirely. Generated code should still be. Tested in the usual way. Finally citations are only helpful when developers look at the cited sources of simply accepting the generated answers without checking.

➢ *Illustrative Usage Scenarios*

Four hypothetical scenarios show how the proposed components can work together. In bug localization exact error messages and class names make lexical retrieval especially helpful while call-graph expansion can add context about the functions linked to the failure. In API discovery, when a developer uses words that differ from those in the code dense retrieval can spot implementations and reranking can raise the best results to the top. On onboarding questions about authentication may need data from files so cross-file retrieval and source citations become vital for letting developers confirm the answer. In code completion inside an IDE response time is a worry so a single retrieval pass may be better than many passes when the first results are enough. These scenarios show that the balance, between retrieval methods, structural context and generation can shift with the task. Therefore a flexible system should let the components be set up to fit each application.

➢ *Benchmark Landscape*

The studies reviewed use several benchmarks, summarized in Table 6, to evaluate different aspects of code retrieval and repository understanding. CodeSearchNet evaluates natural-language code search using Mean Reciprocal Rank (MRR) across six programming languages [1]. RepoEval focuses on repository-level code completion at different levels of granularity, using measures such as exact match, edit similarity, and unit-test pass rate [9]. SWE-bench evaluates systems on real GitHub issues by checking

whether their proposed changes pass the project's tests, while ALCE assesses attributed question answering through citation precision and recall [3]. Although these benchmarks provide useful insights into individual capabilities, they do not fully evaluate the entire process of retrieving repository context, generating an answer, and verifying its correctness within a single private codebase. For this reason, evaluating a complete repository assistant should also include a small, project-specific test set based on realistic developer questions.

Table 6 Benchmarks Used in the Reviewed Studies

Benchmark	Task	Main metrics	Ref.
CodeSearch Net	NL-to-code retrieval, six languages	MRR	[2]
RepoEval	Repository-level completion (line, API, function)	EM, ES, unit-test pass rate	[9]
SWE-bench	Resolving real GitHub issues	Resolved rate (tests pass)	[4]
ALCE	Attributed question answering	Citation recall and precision	[10]

➢ *Relation to Existing Surveys*

Several surveys touch on topics. One broad survey on retrieval-augmented generation divides pipelines into advanced and modular types but focuses mainly on text. Another survey and toolkit for learning on code organizes models, tasks and representations without paying attention to retrieval pipelines. A vision paper on AI in evidence-based software engineering centers on literature reviews. The current review stands in the middle of these approaches. It begins with a query moves through dense retrieval then to syntax-aware chunking and cross-file context and ends with attributed generation. It also includes infrastructure and evaluation aspects that the other surveys do not emphasize. Still this review has its limits. It does not take the place of a study that tests all components under the same protocol. The figures in this review should be seen as signs of trends than, as a strict ranking.

VI. CONCLUSION

This review gathers research on retrieval and AI-based software repository understanding across four main topics: lexical retrieval, dense representations, repository structure and evidence-grounded generation. The results show that adding BM25 to retrieval can help with both exact identifier searches and broader conceptual questions. Rank fusion and reranking can make the results more relevant. Organizing code chunks around AST boundaries keeps code units. Iterative retrieval and generation can add context from across a repository. Evidence-grounded generation lets developers compare AI explanations with sources but reliable attribution is still important. Together these methods give a base, for building repository assistants that provide relevant, context-aware and verifiable answers.

Despite these results some problems still exist. Hybrid search systems can add time to process break down store indexes and work together. Current tests also only look at a programming languages while methods like changing how things are combined using groups of agents and making short summaries need more tests. Future studies should look into adding information without redoing everything combining searches based on what people are looking for and making smaller ways to show how parts of code are connected. Other important areas are learning from diagrams with code making it possible to use these tools on computers with smaller models and creating helpers that check and run the code they make. Solving these issues could make tools for code repositories better easier to use and more trustworthy, in software work.

VII. FUTURE SCOPE AND RESEARCH DIRECTIONS

The reviewed literature points to several directions that we expect to matter for future repository understanding systems.

➢ *Incremental and Streamed Indexing*

Many repository assistants think that parsing, embedding and index construction can be finished once and then reused. Software repositories change a lot with commits, branch switches and rebases. This makes rebuilding the index very costly. Research on retrieval systems shows that encoding collections of passages takes a lot of computing power. Future systems should support updates. Future systems should track changes between commits reprocess the AST subtrees that were affected and update both lexical and vector indexes without rebuilding them completely. This approach could lower maintenance costs. Keep repository information up, to date.

➢ *Multimodal Repository Understanding*

Software repositories often contain more than source code, including architecture diagrams, sequence charts, entity–relationship schemas, API specifications, and infrastructure definitions. However, these resources may be overlooked or reduced to plain text by existing retrieval systems. Future research could explore methods that connect visual diagrams with code structure and use vision-language models to link diagram components to the services that implement them. This could help repository assistants answer broader system-level

questions that cannot be fully resolved by examining source code alone.

➢ *Local and Edge Deployment*

Privacy concerns and the need for faster responses are encouraging developers to run retrieval and generation locally or on on-premises servers. Achieving this requires compact language models, quantized retrievers, prompt compression, and lightweight vector stores that can operate within practical response-time limits. As potential design targets, search could aim for a latency of around 200 milliseconds, while inline code completion could aim to respond within approximately one second.

➢ *Closed-Loop Verification*

Repositories come with a built-in verifier in the form of compilers and unit tests. Agents that retrieve evidence, propose a patch or an explanation, compile and test it in a sandbox, and then use the resulting traces to refine both the query and the output could go beyond passive retrieval. Evaluating such agents calls for benchmarks that score execution outcomes rather than text similarity, as SWE-bench does.

➢ *Adaptive Fusion and Calibrated Confidence*

Fixed fusion strategies do not always fit every type of query. Query-aware weighting, learned fusion methods and calibrated confidence estimates could help retrieval systems adjust to the information being asked for. For questions involving identifiers or function names, retrieval systems could give more weight to lexical retrieval while conceptual questions could benefit more from dense semantic retrieval. Confidence estimation could also help the notice when the retrieved evidence is not enough to support a reliable answer letting the assistant express uncertainty instead of making a potentially misleading answer.

ACKNOWLEDGEMENT

The authors are grateful to their guide, Prof. B. S. Kulkarni, for her guidance and encouragement throughout this work, and to the Department of Computer Engineering at Pune Institute of Computer Technology for its support.

REFERENCES

[1]. H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “CodeSearchNet challenge: Evaluating the state of semantic code search,” arXiv:1909.09436, 2019.

[2]. Y. Wan *et al.*, “Deep learning for code intelligence: Survey, benchmark and toolkit,” *ACM Compute. Survey.*, vol. 56, no. 12, Art. 309, Oct. 2024, doi: 10.1145/3664597.

[3]. P. Lewis *et al.*, “Retrieval-augmented generation for knowledge-intensive NLP tasks,” in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 33, 2020, pp. 9459–9474.

[4]. Y. Gao *et al.*, “Retrieval-augmented generation for large language models: A survey,” arXiv:2312.10997, 2023.

[5]. M. Esposito, A. Janes, D. Taibi, and V. Lenarduzzi, “Generative AI in evidence-based software engineering: A white paper,” arXiv:2407.17440, 2024.

[6]. M. Marcelino, M. Alves, B. Trinkenreich, B. Cartaxo, S. Soares, S. D. J. Barbosa, and M. Kalinowski, “Investigating the use of LLMs for evidence briefings generation in software engineering,” arXiv:2507.15828, 2025.

[7]. X. Li *et al.*, “CodeRetriever: Large-scale contrastive pre-training for code search,” in *Proc. 2022 Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, Dec. 2022, pp. 2898–2910.

[8]. W. Gu, Z. Li, C. Gao, C. Wang, H. Zhang, Z. Xu, and M. R. Lyu, “CRaDLe: Deep code retrieval based on semantic dependency learning,” *Neural Netw.*, vol. 141, pp. 385–394, Sept. 2021, doi: 10.1016/j.neunet.2021.04.019.

[9]. W. Cheng, Y. Wu, and W. Hu, “Dataflow-guided retrieval augmentation for repository-level code completion,” in *Proc. 62nd Annu. Meeting Assoc. Comput. Linguistics (ACL), Vol. 1: Long Papers*, Aug. 2024, pp. 7957–7977.

[10]. S. Zhang, Y. Ding, S. Lian, S. Song, and H. Li, “CodeRAG: Finding relevant and necessary knowledge for retrieval-augmented repository-level code completion,” in *Proc. 2025 Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, Nov. 2025, pp. 23278–23288.

[11]. J. Strich, F. Schneider, I. Nikishina, and C. Biemann, “On improving repository-level code QA for large language models,” in *Proc. 62nd Annu. Meeting ACL, Vol. 4: Student Research Workshop*, Aug. 2024, pp. 209–244.

[12]. W. Gu, Z. Li, C. Gao, C. Wang, H. Zhang, Z. Xu, and M. R. Lyu, “CRaDLe: Deep code retrieval based on semantic dependency learning,” Neural Networks, vol. 141, pp. 385–394, Sept. 2021, doi: 10.1016/j.neunet.2021.04.019.