# Survey and Comparison of Code Repository Understanding and Context Retrieval Techniques

Ratul Vaish

*Electrical and Electronic Engineering, University of Warwick*
`ratul.vaish@live.warwick.ac.uk`

March 2025

**Abstract**

Retrieval-Augmented Generation (RAG) and Agentic AI have been thrust to the forefront of the Artificial Intelligence field. With both being implemented by leading companies in the tech industry. In this project I will evaluate both RAG and Agentic AI on their ability to efficiently and effectively navigate and and retrieve files from large codebases. Utilising the SWE-bench verified dataset to act as a validated and proven input to my algorithms. Furthermore, as code retrieval with SWE-Bench is hard to evaluate I will be implementing BM25 as a baseline approach. The files are publicly available on github via this link.
github.com/RatVaish/3rdYearProject

## 1 Introduction

Context retrieval presents significant challenges in today's information-rich era [31]. With many different databases and repositories all with different structures and representations, this is a major block to true artificial intelligence.

In recent years, Large Language Models (LLMs) have proven themselves pivotal forces in the fields of natural language processing, embodied AI, and AI-generated content [105]. For example, ChatGPT and DeepSeek can be applied as general solvers to problems, following human descriptions of tasks and performing multi-step reasoning where needed [68]. However, LLMs lack the ability to tackle complex problems while maintaining accuracy. Context retrieval is key to feeding LLMs extra information, and thus generating a prompt that enables an accurate and relevant output.

This paper compares three different methods: Agentic AI, Retrieval-Augmented Generation (RAG) and a baseline BM25 approach, to context retrieval from a software engineering perspective. This will use the data available from the SWE-Bench dataset [40] to act as the source to my input informations and the answer set.

Agentic AI leverages LLMs to make decisions, using different tools to interact with a problem and essentially to be able to *"think"* and solve the problem. This new approach shows promise, with much evidence pointing to it being the future of the AI industry.

RAG has also proved to be useful for the development of AI in decision making. With its approach being more computationally heavy, but allowing for links to be made between many different mediums of information.

## 2 Relevant Literature

This section serves as an in-depth analysis into the current field of automated software development covering basic term descriptions and reviewing some of the many code generation approaches currently used in industry. Initially delving into relevant background information and techniques, and then decomposing some of the top approaches to code and patch generation in the current environment.

### 2.1 Basic Background

**SWE-bench** is an evaluation framework consisting of 2,294 software engineering Python problems sourced from popular GitHub repositories [40]. It provides a large and diverse set of problems that reflect real-world software engineering challenges, offering a comprehensive resource for testing and benchmarking code retrieval systems.

The framework is divided into two main datasets: SWE-bench lite and SWE-bench verified. SWE-bench lite contains a subset of 300 problems that are easier to solve, making it suitable for rapid prototyping and testing of basic retrieval algorithms [112]. In contrast, SWE-bench verified contains a subset of 500 problems that are more complex and standard, designed to rigorously evaluate the performance of code retrieval systems in more realistic settings [18].

For this paper, we focus on SWE-bench verified, as it provides a more challenging and representative dataset for assessing the capabilities of advanced code retrieval models, and it contains human-verified solutions for accurate testing and evaluation [7].

**Large Language Models** (LLMs) have significantly advanced the field of automated code retrieval by leveraging deep learning techniques to understand and generate code. Models such as OpenAI's Codex [14] and Google's PaLM [20] have demonstrated strong performance in code-related tasks, including retrieval, completion, and synthesis.

These models employ transformer-based architectures [91], enabling them to capture complex semantic patterns in code. Recent studies have explored fine-tuning LLMs using a code corpus specific to a field, to enhance retrieval accuracy [52]. Furthermore, retrieval-augmented generation (RAG) approaches provide LLMs with external knowledge sources to refine code search results dynamically [51].

Future success of LLMs poses computational challenges, as they require more efficient techniques to enable real-world use [28]. As research progresses, optimizing LLMs for code retrieval remains a crucial area for improving automated software engineering tools.

**GPT-4** has shown considerable promise in the domain of code retrieval, leveraging its deep understanding of natural language and code syntax to retrieve relevant code snippets based on problem descriptions. By generating embeddings for both the input queries and code snippets, GPT-4 can match semantically similar code even when the exact keywords are not present.

This capability is particularly useful in scenarios where traditional keyword-based search methods may fall short. The model's ability to understand both high-level descriptions and specific code context allows it to provide more accurate retrieval results, enhancing productivity in software development [14].

Furthermore, when integrated with advanced techniques like embeddings and program synthesis, GPT-4 facilitates a more efficient and effective search in codebases. As AI-powered tools for code retrieval continue to evolve, GPT-4 stands as a leading example of how large language models can transform code search and retrieval workflows.

**Retrieval-Augmented Generation** (RAG) is an innovative approach that combines information retrieval and generative models to enhance performance in tasks such as code retrieval and generation. The core idea of RAG is to retrieve relevant information from an external knowledge base, such as a code repository, and use it to guide the generation process [51].

This hybrid approach has shown significant improvements over traditional generation methods by grounding the output in real-world data, leading to more accurate and contextually relevant code snippets [118]. When discussing code retrieval, RAG can dynamically refine search results by iteratively referencing external knowledge into the generation process [43].

Recent work has explored the combination of RAG with large-scale pre-trained language models (LLMs) like GPT-3, which further enhances the quality of retrieved code through more fine-grained contextualization [96]. As RAG continues to evolve, optimizing the balance between retrieval and generation remains an active area of research, particularly for resource-constrained environments [53].

**BM25** is a widely used probabilistic information retrieval model based on the term frequency-inverse document frequency (TF-IDF) ranking scheme. It is an extension of the classic Okapi BM model and has been shown to perform well in a variety of text retrieval tasks, including code retrieval [80]. BM25 considers the frequency of query terms within a document, adjusting for document length and account for the fact that after a certain frequency relevance ceases to increase [79].

In the context of code retrieval, BM25 can be used to rank code snippets or files by their relevance to a given query, allowing for efficient retrieval of relevant code from large repositories [25]. It has been applied in conjunction with other methods, such as embedding-based retrieval or deep learning models, to improve performance in tasks like code search [103]. While BM25 remains a strong baseline for traditional search tasks, recent advancements in combining BM25 with more complex retrieval techniques show promising results in the context of larger codebases [106].

**Agentic AI** refers to autonomous systems capable of making decisions and taking actions to achieve a goal with minimal human intervention. In the context of code retrieval, agentic AI frameworks enable automated agents to select appropriate search techniques, refine results, and solve coding problems with minimal human guidance [78]. These systems typically utilise multiple AI methods, such as natural language processing (NLP) and knowledge retrieval, to enhance performance in complex tasks like code understanding, modification, and generation [73]. One of the key aspects of agentic frameworks is the ability to adaptively select between different algorithms or tools built to tackle respective tasks, a feature that has been explored in areas like automated bug fixing [126].

In structured codebases, where code is typically modular and well-documented, agentic AI can employ autonomous systems to navigate, understand, and modify repositories to make more informed decisions when retrieving and generating code [29]. These systems often use a combination of program analysis techniques, such as abstract syntax trees (ASTs) and control flow graphs, to understand code structure and dependencies [35]. One key advantage of agentic AI in structured environments is its ability to intelligently explore the codebase by iterating through various search methods, such as keyword searches and code embeddings, to identify relevant components for a given task [86]. These frameworks incorporate feedback loops where results are iteratively refined, with the system learning from past queries to improve future performance [114].

Such systems can also employ reinforcement learning to optimize their search strategies, refining their decision-making process as they interact with the codebase [32]. Additionally, agentic AI frameworks operating in structured codebases benefit from the ability to reason about both high-level and low-level details, enabling the generation of more precise code modifications [98]. These advancements are essential for automating tasks such as bug fixing and code refactoring in large-scale code repositories. The rise of agentic approaches holds promise for more efficient and scalable solutions in software engineering tasks, paving the way for intelligent systems capable of high-level problem-solving.

An essential feature of agentic AI is its ability to engage in self-criticism, which involves evaluating its own actions, decisions, or performance to identify areas for improvement [4]. Self-criticism in agentic AI enables systems to learn from past experiences and adapt to new situations, thereby enhancing their decision-making capabilities and increasing their effectiveness over time [6]. This process is particularly important in complex environments, such as code retrieval, where AI agents need to assess their ability to retrieve the most relevant information and adjust their approach accordingly. The concept of self-criticism in agentic AI is closely related to reinforcement learning, where agents continuously refine their strategies based on feedback from their environment [69]. However, there are challenges associated with implementing self-criticism, especially in real-world applications, where it is difficult to define objective criteria for performance and feedback may be sparse or noisy [27]. Research into enhancing the self-criticism mechanisms of agentic AI is ongoing, with recent approaches focusing on meta-learning and self-improvement [82]. These approaches aim to enable AI systems to not only critique their own actions but also learn how to improve their critiquing process, resulting in more efficient and intelligent agents [39].

**Context retrieval** is essential to automated code understanding, enabling systems to retrieve relevant information from large software repositories. Effective context retrieval helps in improving code search, bug fixing, and code generation by providing relevant documentation, and code snippets [3].

Traditional keyword-based search methods [70] have been augmented by machine learning approaches, leveraging embeddings to enhance retrieval accuracy [33]. More recent advancements incorporate Large Language Models (LLMs) to understand and process natural language queries or related code snippets [2].

Furthermore, retrieval-augmented generation (RAG) techniques have been employed to improve the efficiency of context retrieval by combining search systems with neural networks [51]. Context-aware retrieval methods implement tools such as Abstract Syntax Trees (ASTs) and graph-based representations to understand the structure and meaning in code [35]. These techniques collectively improve automated software engineering by growing the relevance and accuracy of retrieved code.

**Dependency control** plays a crucial role in modern software systems, where managing interdependencies between code components is essential for maintaining modularity, reusability, and robustness. In the context of code retrieval, dependency control refers to the process of identifying and managing dependencies between various parts of a codebase, such as functions, classes or libraries [90].

Effective dependency management allows retrieval systems to better understand the structure and behaviour of code, facilitating more accurate searches within codebases [76]. Several techniques have been proposed to model code dependencies, including static analysis tools that generate program dependence graphs (PDGs) and control flow graphs (CFGs), which capture the relationships between code elements [62].

Additionally, dependency control is crucial when modifying code, as it helps ensure that changes do not break existing functionality [23]. In code retrieval systems, it allows the agent to recognize how different parts of a codebase interact, improving search accuracy [64]. Recent approaches also incorporate machine learning techniques to predict dependency relationships, enabling more adaptive and intelligent handling of complex codebases [100].

## 2.2 Commonly Implemented Techniques

**Keyword search** is one of the most common and straightforward techniques used in code retrieval systems. It involves searching for specific terms or phrases within the codebase that match the user's query. Keyword-based approaches are efficient, easy to implement, and often serve as a base search method for code retrieval [10].

In code search, the system typically indexes the source code files, allowing for fast retrieval of files or code snippets containing the queries keywords. However, this approach has limitations, especially when dealing with complex queries or when searching for semantically related code. Keywords fails to capture the meaning or structure of code, which in turn leads to false positives or missed results [36]. To overcome these issues, keyword search can be enhanced with techniques such as stemming, which reduces words to their base form, and synonym expansion, which broadens the search scope by including related terms [83].

Additionally, hybrid methods have been proposed that combine keyword search with other techniques, such as abstract syntax tree (AST) matching or semantic analysis, to improve the relevance of the results [17]. These hybrid systems aim to balance the speed and simplicity of keyword search with the accuracy of more advanced search methods. Despite the advancements, keyword search remains a fundamental technique due to its simplicity and effectiveness in many use cases, particularly when dealing with simple well defined problems [16].

**Natural language analysis** plays a crucial role in code retrieval systems, particularly when bridging the gap between human-readable descriptions and machine-understandable code. In code retrieval, natural language analysis is used to process and interpret problem statements expressed in natural language and convert them into structured queries used to retrieve relevant code snippets [63].

This process often involves techniques such as tokenization and named entity recognition to identify key elements in the text, which are then related to corresponding components or functions [108]. More advanced natural language processing (NLP) techniques, such as semantic parsing and embedding-based approaches, have shown to significantly improve retrieval performance by providing deeper understanding of the underlying intent behind user queries [22].

In recent years, transformer-based models like BERT [22] and GPT [75] have demonstrated state-of-the-art performance in understanding and generating natural language, enabling more sophisticated querying and contextual search within large codebases [57]. The integration of NLP techniques in code retrieval systems allows for the translation of ambiguous or complex queries into precise code search, improving the accuracy and efficiency of retrieval [34]. However, challenges remain in adapting these NLP models to handle the unique semantic characteristics of programming languages [77].

**Iterative search mechanisms** are essential for improving the accuracy and efficiency of code retrieval systems. These mechanisms operate by refining the search results over multiple iterations, using feedback from previous steps to progressively narrow down the scope of relevant code. In the context of code retrieval, iterative search techniques often involve the use of heuristics, or relevance feedback which adjusts the search query or ranking of results based on the user's needs or previous results [11]. One popular technique is query expansion, which enhances the search query by adding terms or concepts derived from the retrieved documents, helping to improve the precision of results [125].

Iterative search can also be enhanced through techniques like active learning, where the system actively queries the user for feedback on uncertain results, further refining the retrieval process [67]. In code retrieval, iterative mechanisms have been shown to be particularly effective when integrated with more sophisticated models such as neural networks, where the model refines its predictions and embeddings after each iteration, improving the overall search process [56]. These iterative techniques allow the system to adapt to the user's specific query and context, making the retrieval process more dynamic and responsive especially when faced with more complex coding tasks [58]. However, challenges remain in balancing efficiency and accuracy, especially when handling large-scale codebases [26].

**Abstract Syntax Trees** (ASTs) are a widely used technique in code retrieval systems for representing the syntactic structure of code. An AST is a tree-like data structure that reflects the hierarchical and syntactic structure of a programming language's source code, where each node represents a construct such as an expression, statement, or function [72].

By analysing the structure of code using ASTs, retrieval systems can identify patterns, detect structural similarities, and capture the underlying logic of the code, enabling more precise and context-aware code search [91]. ASTs are particularly effective in dealing with the challenges of code retrieval because they abstract away language-specific syntactic details while preserving the logical structure, making them robust to variations in code formatting, naming conventions, or even minor code refactoring [66].

Additionally, ASTs enable the use of symbolic execution and other static analysis techniques, which can enhance the precision of code retrieval by focusing on the functional aspects of code [74]. Recent advancements have leveraged deep learning techniques, such as neural AST embeddings, to enhance the use of ASTs in code retrieval tasks. These methods generate low-dimensional vector representations of ASTs that can be used for similarity matching, enabling more efficient and accurate retrieval of relevant code snippets [97]. Despite their effectiveness, processing large-scale ASTs efficiently is a challenge, as well as integrating AST-based techniques with other retrieval methods such as natural language processing [57].

**Function call graphs** (FCGs) are a powerful representation of software systems, capturing the relationships between functions and their implementation. In the context of code retrieval, FCGs serve as a valuable tool for understanding the structure and behaviour of code by mapping out which functions call other functions and how data flows between them. This graph-based approach is particularly useful for analysing complex codebases, where traditional keyword-based or syntactic search methods may fail to capture the deeper relationships [12].

By focusing on the interactions between functions, FCGs enable more accurate code retrieval, as they allow for the identification of relevant code snippets that share similar functional behaviours, even if they do not share keywords or structure [109]. Function call graphs can be constructed through static analysis, where the code is analysed without execution, or dynamic analysis, which tracks function calls during program execution [30]. One significant advantage of using FCGs in code retrieval is their ability to represent and leverage higher-level abstractions, facilitating the retrieval of code snippets that may be contextually related or exhibit similar functionality [111].

Furthermore, FCGs can be combined with other retrieval techniques, such as abstract syntax trees (ASTs) or semantic embeddings, to improve the precision and recall of the search results [55]. While the construction of function call graphs can be computationally intensive, especially for large and dynamic codebases, their ability to enhance retrieval accuracy makes them a valuable tool in modern code search systems.

**Monte Carlo Tree Search** (MCTS) is a popular search algorithm used in artificial intelligence, especially in decision-making processes such as game playing and planning. MCTS has gained attention for its application in code retrieval due to its ability to handle large search spaces and evaluate multiple possible solutions through simulation-based approaches. In the context of code retrieval, MCTS can be employed to explore different paths for retrieving relevant code snippets, simulating this process to assess the quality of different search strategies

[21].

The algorithm builds a search tree incrementally, selecting nodes (representing code fragments) based on their potential to improve the retrieval results. At each iteration, the search tree is expanded by selecting the most promising nodes, simulating their outcomes, and returning the results to update the search strategy [19]. One of the strengths of MCTS is its ability to balance exploration and exploitation, making it well-suited for code retrieval in dynamic and large codebases [9].

Additionally, MCTS can be combined with other retrieval techniques, such as semantic analysis or keyword matching, to refine the search process and improve the accuracy of retrieved code snippets [102]. Despite its strengths, MCTS can be computationally expensive, especially when dealing with highly complex and large codebases or highly specific retrieval tasks [84]. Nevertheless, the adaptability and potential for improving search outcomes make MCTS a valuable tool in the field of automated code retrieval.

**Fault localization** is a critical technique in code retrieval systems, aimed at identifying the location of faults or errors in code. This process is vital for efficiently diagnosing issues in large codebases, as it helps to pinpoint the areas of code that are most likely responsible for a failure. In the context of code retrieval, fault localization techniques typically leverage both dynamic and static analysis of the code. Dynamic analysis involves executing the code and tracking program behaviours, while static analysis focuses on examining the source code's structure and dependencies without executing it [42].

Among the most common techniques for fault localization are spectrum-based fault localization (SBFL) and symbolic execution. SBFL uses program coverage data, such as which lines of code were executed during a test case, to rank code components based on their likelihood of containing faults [124].

Symbolic execution, on the other hand, analyses the execution paths of the code under various conditions, helping to identify differences between expected and actual behaviour [104].

These methods are particularly useful in large software systems, where manually tracing errors can be time-intensive and result in frequent errors. Recently, machine learning models have also been applied to fault localization, with deep learning techniques being used to predict faulty code sections based on patterns gleamed from bug data [87]. These approaches can enhance the accuracy of fault localization by incorporating contextual information from the code and test results, thus improving the overall debugging process. Despite their effectiveness, challenges remain in dealing with complex software systems, where multiple faults can complicate the localization process [38].

## 2.3 Current Solutions

When working with the SWE-Bench dataset [40], there is a lot of documentation available on multiple different approaches to solve the set of problems. While these usually focus on code generation, every approach has a slightly different take on context retrieval, which could heavily impact their final results.

### 2.3.1 Auto Coders

The rapid advancement of LLMs has led to significant developments in automated coding systems, often referred to as "auto coders". Auto Coders primarily follow a modular, iterative, and self-improving approach. Their goal is to enhance software development by automating code retrieval, understanding and modification [37]. A critical component of their functionality lies in their ability to decompose problems and retrieve relevant context effectively.

Problem decomposition is essential for auto coders to accurately identify and address issues within a project.*AutoCodeRover* [119] analyses GitHub issues using natural language descriptions to isolate relevant program components, leveraging abstract syntax trees (ASTs) to pinpoint relevant classes and methods. Specifically, through the application of a set of multiple retrieval APIs that search for classes and functions with embedded subclasses and functions. This structured representation allows a system to efficiently refine its problem-solving input.

*SpecRover* [81] feeds the GitHub issue statement into a reproducer agent that reproduces the described program fault. This is then passed as context along with the GitHub issue statement and the GitHub project codebase, into a context retrieval agent. Similarly to *AutoCodeRover*, this is performed iteratively to ensure the retrieved context is correct.

The approach taken in *"Automated Repair of Programs from Large Language Models"* [24] categorizes errors in code generated by LLMs, employing a systematic fault pattern identification process that guides repair mechanisms.

Program dependence graphs (PDGs) and control flow analysis have also been employed to refine problem decomposition strategies, equipping automated systems to better capture interdependencies within codebases [85, 41]. These methodologies highlight how modern auto coders strategically process a problem, improving efficiency and accuracy of context retrieval and code generation.

Context retrieval is another fundamental challenge in auto coders, as accurate context retrieval of relevant code snippets and dependencies significantly improves code generation and program modification outcomes.

*AutoCodeRover* [119] employs an iterative search mechanism using ASTs, searching for a location in the GitHub project codebase that causes the problem. This then returns a filename, relevant function names, and any other relevant context to a final code generation agent. This approach ensures dependencies remain constant and allows for efficient generation.

*SpecRover* [81] enhances this process with a reviewer agent that evaluates the proposed patch generated after using a context retrieval agent that operates similar to *AutoCodeRover*. This offers explanations and reliability assessments to improve overall accuracy and transparency.

Additionally, the approach in *"Automated Repair of Programs from Large Language Models"* [24] utilizes statistical fault localisation to pinpoint bug locations, and then proceeds to integrate contextual information into the repair process. ‘

Beyond these approaches, some systems employ deep learning techniques to further automate context retrieval. Models such as CodeBERT [120] and GraphCodeBERT [65] have been integrated into auto coder workflows, enabling more accurate understanding and retrieval of code contexts. Furthermore, some approaches leverage reinforcement learning techniques to refine the problem-solving process over time, allowing auto coders to improve their efficiency as they learn from previous iterations [48].

Auto coders are rapidly transforming software engineering by automating the problem-solving and repair process. A combination of effective problem decomposition methods and advanced context retrieval methods enable these systems to efficiently address programming issues.

The studies reviewed highlight how techniques such as ASTs, fault localisation, and iterative search processes contribute to growing sophistication in auto coders. Future research that expands on these methods or similar approaches should continue to refine the efficiency and accuracy of auto coders, thus enabling them to adapt to the rapidly growing complex software environment.

### 2.3.2 Top SWE-Bench Solvers

While auto coders represent significant progress in automated code generation, they aren't the only approaches to dominate the SWE-Bench leader-boards [40]. The most successful approaches take a more human-like strategy, equipping agents with a diverse set of tools that they can dynamically choose to use. By mimicking the problem-solving mindset of a software engineer, these methods often achieve higher solve rates.

***GRU*** [110] is one such approach, which follows a broad step-by-step approach to problem decomposition, context retrieval and code generation. Initially, it uses keyword search on the problem description, allowing it to narrow down potentially relevant functions in the problem. Next, utilising this input, it searches the project codebase for files relevant to the problem description, reading what it finds interesting and saving all findings. This eventually leads to it generating a file change plan after finding more context based on what it has already received, and then finally, it generates a patch to the problem.

This is a novel approach that makes the most of the improved intelligence of LLMs such as GPT-4 to solve problems. However, in their paper [110], they noted several possible improvements that if implemented could further improve accuracy. These include: implementing a synthetic RAG search instead of a keyword search for a massive boost in hit rate, building an initial knowledge base of the current project for *GRU* to use, and adding multimodal capabilities to glean further information from graphs and images provided in the project files [59], [121].

***Abante AI*** [1] is another optimistic approach that follows a simple iterative approach to code generation. *Abante AI* feeds the original GitHub issue into a context gathering agent, which in turn feeds into a plan and edit agent, the results from this are then tested and reviewed, and the program determines whether to fetch more context, re-generate the plan or submit the solution.

In the Gather Context Stage *Abante AI* determines what information is useful from the codebase using a simple unique approach called Library context. This requires no preprocessing of the codebase and takes only a few seconds to compute, utilizing parallel calls to the model to select important files [50], [44].

Next, the Plan and edit stage, during this *Abante AI* makes a plan and then implements it as a series of edits to files using an LLM. This step also writes tests for its changes which run in parallel with previous tests and feed directly into the final stage Test and Review, as context.

Finally, in the Test and Review segment, *Abante AI* implements simple step-by-step logic. Submit, if the model is confident in its produced solution, add more files to context, make additional edits, or reset all edits made and start over with a note from the previous iteration acting as a note as to what went wrong in the initial approach.

This was run with a maximum of 3 iterations and if there were no successful results submitted, the system is reset and rerun. This is in-turn allowed to happen for a maximum of 8 attempts and is considered the Pass@1 Score. If all 8 attempts submit no solutions, a heuristic approach is employed to choose the best approach to submit for evaluation. Overall, it was found that the system solved 102 benchmark instances in 3 or fewer attempts, beating the previous state of the art available [45].

***SWE-Search*** [5] is a multi-agent system designed to tackle complex software engineering tasks through emulation of the sophisticated, iterative workflows of human software engineers. Using Monte Carlo Tree Search (MCTS) for planning, a Value agent for utility estimation and feedback, and a Discriminator agent for final deductive decision making, *SWE-Search* provides an adaptive framework capable of solving real-world software engineering challenges.

*SWE-Search* can be broken down into four primary components:

- *SWE-Search Framework and Action Agent*: This builds on the *moatless-tools* [71] framework, operating in a dynamic code environment with a flexible state-space and a git-like commit tree structure. This facilitates efficient backtracking to previous states, enabling the Action agent to explore diverse solution trajectories. Adapting state-space enhances the system's ability to exploit the MCTS effectively.

- *Value Agent*: This is an LLM-based value function, that outputs a value to approximate the utility for each observation and an explanation in natural language. This can be leveraged to improve subsequent actions from parent nodes, enabling iterative self improvement [117].

- *Search Algorithm*: The core of *SWE-Search*'s search strategy is based on Monte Carlo Tree Search (MCTS) which employs a heuristic-based selection process. This MCTS algorithm effectively balances exploration and exploitation, helping agents explore a diverse set of solutions and determine the most promising strategy.

- *Discriminator Agent*: This is the final stage of *SWE-Search*, it evaluates the solutions generated by the search process. This agent engages in a structured debate in which multiple agents argue for or against the proposed solutions, leading to more rigorously justified final decisions.

***Factory Code Droid*** [89] is another model approach, this automates software development using task decomposition, multi-model sampling, and self-critique to iteratively refine solutions. It retrieved relevant code with HyperCode and ByteRank and validates outputs through testing and optimisation.

In order to perform complex software development, AI systems need to be able to decompose high-level problems into more manageable subtasks. *Factory Code Droid* translates these subtasks into action space and reasons around optimal trajectories [60]. To achieve this optimally, several techniques were developed, enabling the systems to simulate decisions, perform self-criticism, and reflect on real and imagined decisions.

*Factory Code Droid* has access to many essential tools for software development, with the general approach being that "If a human has access to a tool, so should the system." Some relevant examples include debuggers, static analysers, and version control systems [46]. This grounds *Factory Code Droid* in the environment of a software engineer and ensures the same feedback and iterative loops used by human software developers.

While most agentic systems enter a codebase with no prior knowledge and build understanding from scratch, *Factory Code Droid* has developed a system called HyperCode. This constructs a multi-resolution representation of any given engineering system and allows a system to synthesize deep codebase understanding [92]. This involves the autonomous construction of explicit and implicit relationships within low-level data, extracting relevant insights at different levels of abstraction. These insights are leveraged by a retrieval algorithm, ByteRank, to retrieve relevant information.

Another advantage of *Factory Code Droid* is that it utilises different LLMs for different subtasks, as this provides multiple trajectories for a given task. These are validated using tests, and the optimal solution is selected [60]. The use of different models ensures diversity and robustness in the final results.

***OpenHands*** [94] approaches AI agent development similarly; from the perspective of a human software developer. With *OpenHands* being capable of writing code, executing commands in a sandboxed environment, and browsing the web to gather information.

Their platform comprises several key components:

- *Agent Implementation*: This creates agents that utilise LLMs to generate code and make decisions, designed to mimic the problem-solving approaches of human software engineering [8].

- *Sandboxed Code Execution*: To ensure safety and correct implementation, the sandboxed environment creates a space for agents to execute code. Isolation enables the agent to test and debug its code without affecting the host system [123].
- *Command-Line Interaction*: Enables agent interactions with the command-line interface, allowing the agent to perform tasks such as file manipulation, running scripts, and managing processes, thus resembling human software developers.
- *Web Browsing Capabilities*: This equips agents with the ability to search for documentation, gather information, and stay updated with the latest developments in relation to their tasks.
- *Coordination Between Multiple Agents*: This facilitates collaboration between multiple agents for a complex task. This creates the opportunity to interface multiple specialised agents to tackle the same problem in conjunction [49].

By integrating these components, it allows for AI agents that can autonomously handle software development tasks, leveraging tools and strategies commonly utilised by human software engineers. *OpenHands* enhances the agent's problem-solving ability and generates potential for more effective real-world applications.

***SIMA*** [54] employs an algorithm called *Agent Forest*, which follows a two-phase implementation: sampling and voting. A problem query feeds into a range of LLM agents, where each outputs a unique answer. Then, through majority voting, these answers are assessed to ascertain the best approach [13].

Their approach to prompt engineering considers multiple comprehensive experiments, evaluating Chain-of-Thought prompting (CoT), Zero-Shot Chain-of-Thought prompting (Zero-Shot CoT), and more sophisticated methods such as Solo Performance Prompting (SPP). These methods are first used with a single LLM query, and then increase the number of queries while employing majority voting to determine the most consistent answer [95].

*Agent Forest* also leverages collaboration between multiple LLM agents to simulate reasoning. It utilizes LLM-Debate and self-reflection as methods to iteratively generate multiple samples. This is applied in parallel with prompt engineering methods to produce the final answer [115].

***SWE-agent*** [107] attempts to provide AI agents with the same set of complex tools that are available to human software developers, aiming to create an interface similar to applications like VSCode but for agents to use.

They found that a few design principles were especially important when building agent-computer interfaces (ACIs):

- *Actions should be simple and easy to understand.* Some bash commands have documentation that includes dozens of options; simple commands with few options and concise documentation are easier for agents to implement. This reduces the need to fine-tune and provide demonstrations at a later step [47].

- *Actions should be compact and efficient.* Important operations such as file navigation and editing should be consolidated into as few actions as possible. A poor design would implement multiple actions composed of higher-order operations, while an efficient design would make meaningful progress with a single action [101].

- *Environment feedback should be concise and informative.* High-quality feedback should contain substantive information about the current environment state and the effect of the agent's actions without unnecessary information [122].

- *Guardrails to mitigate error propagation.* The goal is to build agents to mimic human software developers, and as such, they make mistakes that they can struggle to recover from. Guardrails, such as code syntax validation, can help agents recognize and quickly correct errors [107].

While the above outlines general guidelines for designing an ACI, their approach also details various steps and methods used. In particular, their search and navigation and file viewer functions are notable.

*Search and navigation.* A common approach to navigating codebases is finding useful terms such as files, functions, or class definitions mentioned in an issue. *SWE-agent* approaches this by introducing *find_file*, *search_file*, and *search_dir* functions, which submit a summary of search results when searching for filenames and strings within files or directories. This is further supported by suppression of verbose results, a limit of 50 result returns, and reiteration with a more specific query if this maximum isn't satisfied [101].

*File viewer.* The file viewer function presents the agent with a window of at most 100 lines of the file it wanted to view. This window can be moved with commands such as *scroll_down* and *scroll_up*, and there is a *goto* command to find specific content. This allows the agent to view the entire file in smaller chunks that enable it to understand more efficiently [47].

This approach to problem decomposition and code retrieval enables an LLM to perform at its best ability while avoiding confusion, ensuring accurate and relevant results. However, it is highly dependent on the intelligence of the LLM and could be very resource-intensive [122].

***MAGIS*** [88] opts for a framework that incorporates multiple LLM-based agents working collaboratively. It comprises four unique and highly specialized agents that operate similarly to a team of software developers:

1. *Manager.* The manager is pivotal to planning. Conventionally, managers decompose an issue into tasks and allocate these tasks to members of a preassigned team. However, in *MAGIS*, the Manager can decompose a problem and design developer agents to form a team, massively improving flexibility, adaptability, and efficiency [15].

2. *Repository Custodian.* The repository custodian agent is tasked with locating files relevant to the issue. However, unlike humans who can browse all repository files, LLMs are constrained by their high computational cost in iterative queries and their performance degradation given a long context input [116].

3. *Developer.* While numerous agents already exist that are capable of generating code, their proficiency in modifying code is comparatively lacking. To address this, *MAGIS*'s Developer agent decomposes code modifications into sub-operations that include code generation, allowing them to leverage existing technology for more effective code modification [61].

4. *QA Engineer.* In software development, quality assurance engineers maintain software quality through code review. However, despite their importance, code review practices are often overlooked. To overcome potential delays caused by this, *MAGIS* implements its QA Engineer agent in parallel with its Developer agent to enhance software quality [88].

This approach to code generation and modification allows for better quality control and more efficient production while maintaining relatively low computational costs.

***Agentless*** [99] is an approach that addresses the limitations of current LLMs by removing decision-making from the LLM itself. Instead, LLMs are implemented strictly as tools to process problems and generate code.

The current limitations of LLMs result in the following restrictions in agent-based approaches:

- *Complex Tool Usage/Design.* Agent-based approaches use an abstraction layer to map actions to API calls, enabling tool usage. However, designing precise input/output formats is crucial, as errors can lead to incorrect tool usage, reducing performance and increasing costs due to unnecessary LLM queries. This issue is amplified in iterative processes, where each action depends on previous steps [93].

- *Lack of Control in Decision Planning.* Agent-based approaches not only use tools but also handle decision-making, determining actions based on prior steps and environmental feedback. However, minimal validation can lead to confusion and suboptimal exploration due to the vast action space. Complex issues may require 30-40 turns, making it challenging to trace and debug incorrect decisions [99].

- *Limited Ability to Self-Reflect.* Existing agents lack strong self-reflection capabilities, making it difficult for them to filter out irrelevant or incorrect information. As a result, errors can propagate through subsequent decisions, leading to amplified mistakes and reduced performance [113].

*Agentless* adheres to a simple three-phase process: localization, repair, and patch validation. During localization, *Agentless* employs a hierarchical process to pinpoint faults, narrowing them down from specific files to relevant classes and functions, and finally to precise edit locations. The localization process utilizes both LLM-based fault detection and classical information-retrieval-based localization [99].

During the repair phase, *Agentless* generates multiple candidate patches for the localized edit locations, while simultaneously producing reproduction tests that replicate the original fault. These tests assist in candidate patch selection. Finally, all remaining patches are re-ranked, and one is selected as the final solution patch.

The *Agentless* method can be condensed into a simple 7-step process:

1. Use embedding-based retrieval to fetch code snippets from the issue description and feed them into an LLM-based algorithm that ranks the top $N$ most suspicious files in the tree-like structure of the project files.
2. This provides a list of suspicious files, but not all will be relevant. Another LLM analyzes the skeleton of each file (declaration headers, classes, and functions) and identifies specific classes and functions of importance.
3. The complete code context is provided to the LLM, which then finalizes a smaller set of edit locations.
4. In the patch validation phase, the LLM attempts to recreate the issue by generating multiple reproduction tests.
5. The optimal solution is selected based on execution results.
6. *Agentless* uses the reproduction test along with existing regression tests for patch ranking and selection.
7. Finally, *Agentless* selects the top-ranked patch as the final patch for submission.

This unique approach deliberately avoids agent integration, leveraging LLMs to perform detailed tasks rather than autonomously deciding future operations. As a result, *Agentless* provides a streamlined and easily understandable design while addressing the previously listed limitations of LLM-based agent systems.

Below is a survey comparing all of the different approaches we have discussed previously.

Table 1: Summary of Current SWE-Bench Patch Generation Approaches

| Approach | Methodology | Model Used | Key Innovation |
|---|---|---|---|
| AutoCodeRover [119] | Iterative retrieval-augmented patch generation | CodeT5 | Multi-step refinement with semantic validation |

*Continued on next page*

*Continued from previous page*

| Approach | Methodology | Model Used | Key Innovation |
|---|---|---|---|
| SpecRover [81] | Specification-guided program repair | GPT-4 | Formal verification for correctness |
| Automated Repair of Programs from Large Language Models [24] | Few-shot learning with LLM feedback loop | GPT-3.5 | Self-supervised error correction |
| GRU [110] | Reinforcement learning-based patch synthesis | Proprietary RNN | Adaptive patch selection |
| Abante AI [1] | Hybrid symbolic-ML repair | GPT-4 | Context-aware repair strategy |
| SWE Search [5] | Retrieval-augmented code transformation | BERT | Efficient bug-fix retrieval |
| Code Droid [89] | LLM-based automated patching | PaLM | Fine-tuned domain-specific model |
| OpenHands [94] | Human-in-the-loop repair | GPT-4 | Interactive debugging system |
| SIMA [54] | Semantic-aware mutation analysis | CodeBERT | AST-based repair validation |
| SWE-Agent [107] | Multi-agent system for repair | GPT-4 | Collaborative patch generation |
| MAGIS [88] | Graph-based program repair | GraphCodeBERT | Leveraging control flow graphs |
| Agentless [99] | Fully self-supervised patching | GPT-3.5 | Agent-free repair pipeline |

The surveyed approaches exhibit a diverse set of methodologies for SWE-Bench patch generation. Several key trends emerge from the comparison in Table 1:

- *Retrieval-Augmented vs. Model-Centric*: Some approaches, such as AutoCodeRover [119] and SWE Search [5], focus on retrieval-based patching, leveraging previously solved examples. Others, such as MAGIS [88], employ model-driven techniques without explicit retrieval.

- *Symbolic vs. Neural Approaches*: Traditional symbolic reasoning (Abante AI [1]) contrasts with neural-based LLM models (e.g., Code Droid [89]), highlighting different strengths in correctness versus adaptability.

- *Human-involved Strategies*: OpenHands [94] integrates human feedback, differing from purely autonomous solutions such as Agentless [99].

- *Graph-Based Methods*: Graph structures are gaining traction, as seen in MAGIS [88], which incorporates program dependence graphs for enhanced patch reasoning.

- *Self-Supervised and Adaptive Learning*: Several models, such as GRU [110] and SIMA [54], rely on reinforcement learning or semantic validation, which helps refine patches iteratively.

**In summary**, while each of these top SWE-Bench approaches [40] employs a variety of different techniques and functions, their overall aim and direction is clear. The future of AI agents lies in the ability to approach problems and prompts as a human software developer would. Promising approaches tend to provide agents with a comprehensive and full set of tools to utilize as they deem necessary, and also implement multiple agents in collaboration.

Future research could integrate the strengths of all these methods to provide improved patch quality.

# 3 Methodology

## 3.1 Approach

This study compares three distinct code retrieval methods to assess their effectiveness in retrieving relevant code snippets from the SWE-Bench dataset:

- **BM25**

- **Retrieval-Augmented Generation (RAG)**

- **Agentic AI**

The objective is to determine the most effective approach by evaluating each method across multiple performance metrics, including retrieval accuracy, computational efficiency, and cost.

### 3.1.1 Data Processing

For this project I am working with data from the *SWE-Bench Verified* dataset, this is advantageous to working with SWE-Bench Full or SWE-Bench Lite for a few simple reasons:

1. **High-Quality Ground Truth**. SWE-Bench Verified ensures that that ground truth solutions are verified manually, heavily reducing noise from incorrect or incomplete issue-fix pairs, and further increasing its reliability.

2. **Better Alignment with Real Fixes**. Unlike SWE-Bench Full which contains all potential issues and fixes, SWE-Bench Verified provides a curated subset which are confirmed to be correct and relevant.

3. **Balanced Dataset Size**. The Full dataset is large, potentially containing redundant and irrelevant information, making retrieval less efficient, while Lite is smaller but lacks coverage. Verified provides a middle ground, ensuring completeness without any excess noise.

4. **More Reliable for Model Evaluation**. Since Verified contains reviewed fixes, using it as a benchmark ensures models learn from clean data, leading to better performance metrics.

5. **Higher Precision in Code Retrieval**. As this project aims to find and retrieve the most relevant filename to a problem. Since Verified is filtered for accuracy, it is the most applicable to reduce incorrect retrieval.

The sections in this database necessary for this project are: *problem_statement*, *repo*, *base_commit*, and *patch*, with each playing a critical role in all three code retrieval implementations.

Using these columns from SWE-Bench Verified, I created my own database which contains the *problem_statement*, *github_api_url* and *patch_files*.

I then further processed this data to make it applicable for each application; with the GitHub API URL being used to extract filenames and code context from a GitHub repository at the specified commit hash, the *problem_statement* being filtered for keywords and preprocessed for implementation, and the *patch_files* being the filenames retrieved from the original *patch* column.

### 3.1.2 BM25

BM25 is a simple method that ranks embedded files in a codebase, and you can implement it as a retrieval method by making it retrieve the top ranked files from the codebase based on an imbedded input problem statement.

In my approach I utilise the Microsoft-CodeBERT embedding model to create embeddings of the filenames and problem statement. Constructing a BM25 corpus using the embeddings of the extracted GitHub repository filenames. This is then queries with a processed problem statement embedding and returns the top relevant filename from the corpus, along with the *patch_files* extracted from the SWE-bench dataset as was explained in the previous section.

This is a very simple method and that's why it acts as the baseline comparison point for both RAG and my Agentic approach.

### 3.1.3 RAG

RAG is a very powerful and well known method, and as such acts as a good comparison point to my agentic approach

Initially my approach to RAG was unorthodox, following a less computationally heavy process. My final approach to RAG was more traditional, embedding an entire code repositories information and similarity searching it against an embedding of the problem statement, created with the same function and model. This would then produce one filename which was considered the most relevant file after the similarity search. Leveraging, FAISS's similarity search to rapidly process the large amount of extracted data from the GitHub tarball URL, and Microsoft-CodeBERT as the embedding model for both code files and the problem statement.

This approach ensures no information is lost, and thus the found file is the most relevant to the problem statement. However, the lack of pre-processing in both the problem statement and code repository cause this approach to be very computationally intensive.

### 3.1.4 Agentic Approach

As became well established in the **Relevant Literature** section, some of the top approaches to agentic code generation approach the problem by trying to replicate the actions and tools employed by a human software engineer. In similar fashion the Code Retrieval Agent I developed for this paper attempts to equip an Agentic AI with necessary tools to decompose a problem, and search a codebase for relevant filenames.

Initially I was set on creating my Agentic AI as a single executable python file. However, in the end I decided upon a modular approach as it offered several benefits over the original approach:

- **Maintainability**. Breaking code into smaller, more manageable modules results in simpler debugging and testing without compromising the whole system.
- **Reusability**. You can reuse modules across different project or within the same project, meaning I could apply this agent to other databases such as SWE-Bench Lite and Full.
- **Scalability**. The modular code allowed me to add new features such as different search methods and problem decomposition techniques with relative ease.
- **Easier Testing**. As I wrote and applied pytest cases for all executable files, this approach ensured this was a simple process.

My Agentic Retrieval repository is structured as follows:

```
Agentic_Retrieval/
    |--- main.py
    |--- agent.py
    |--- data_handler.py
    |--- search_tools.py
    |--- config.py
    |--- keys.env
    |--- cleaned_data.csv
    `--- tests/
        |--- test_agent.py
        |--- test_data_handler.py
        `--- test_search_tools.py
```

This approach creates problem decomposition tools and context retrieval tools in the *data_handler.py* file, repository search tools in the *search_tools.py* file, and the agentic logic in the *agent.py* file. These tools are then tested using *pytest* and *pytest.parametrize* in the ***tests*** subdirectory.

The files *keys.env* and *config.py* exist to store any personal API tokens invoked and ensure that they are loaded correctly. While *cleaned_data.csv* is the data input into the system, and is a direct output of the data processing section of my project code.

#### 3.1.5 Retrieval Evaluation

Finally, I evaluate the results retrieved from the different approaches. There are a few approaches I can take: *precision*, *recall*, *execution time* and *cost*. For this project the main metric of focus is the *precision* which is determined by the following equation.

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives} \tag{1}$$

And both *execution time* and *cost* are used as secondary metrics. To properly evaluate the results.

To evaluate this I run code that processes the results.csv file. This returns the number of true positives, false positives, and false negatives. These are then applied using equation 1.

## 3.2 Experimental Setup

### 3.2.1 Datasets

I evaluate all three approaches against the SWE-Bench [40] Verified dataset, to properly test their ability to solve real-world software engineering issues. SWE-Bench Verified is a human validated subsection of the SWE-Bench dataset containing 500 self-contained problems, each with confirmed patch solutions. Due to computational limitations, RAG was evaluated with a randomly selected subset of this dataset, containing only 10% of the data points.

### 3.2.2 Implementation

**Data Processing**. While the raw SWE-Bench [40] data is applied slightly differently in each of the approaches. I have boiled down the approaches in all of the methods into a few key techniques.

Firstly, as was described previously, the SWE-bench data is processed into a second dataframe that contains *problem_statement*, *github_api_url*, and *patch_files*. The *problem_statement* is composed of a natural language description, code snippets, and examples, while the *github_api_url* is a single URL, and the *patch_files* is a set of one or more filenames where the problem actually lies.

All approaches process the *problem_statement* with a similar method. They apply a preprocessing algorithm that uses regression patterns to identify the different sections in the original problem statement. It then removes the examples section that may confuse any retrieval algorithms and creates a string containing just the natural language description of the problem and any relevant code snippets. The agentic retriever has access to *fetch_problem_keywords* which uses spaCy's natural language search to extract relevant keywords, and *extract_function_name* which uses a regression pattern to find any function names and class names present in the problem statement.

The *github_api_url* is used by all the approaches to retrieve either a list of filenames or a dictionary containing filenames and their contents. This is done by retrieving a GitHub repository tarball using the given URL, then extracting the tar files contents, storing them as initially mentioned.

The *patch_files* do not need to be processed and are used directly in the Retrieval Evaluation section. While missing values in any of the data processing functions print a relevant error message and return an empty value, essentially skipping them.

**BM25**. My approach to BM25, implements the *preprocess_problem* function and the *fetch_filenames* function mentioned previously. It uses these, in conjunction with a Microsoft-CodeBERT embedding model, to create a corpus for the BM25 search.

It then implements a function *retrieve_top_filename*, that tokenises the problem statement and uses a BM25 search to retrieve the most relevant filename from the corpus. This is run row-wise with the results being saved to a .csv file for use in the Retrieval Evaluation stage.

**RAG**. To implement RAG, I employ the *extract_github_files* function, and *preprocess_problem*. I have also created a *get_embedding* function which leverages the Microsoft-CodeBERT tokeniser and model to create embeddings for any input string.

Furthermore, I created a *similarity_search* function that takes an input problem statement and the relevant retrieved GitHub files, creates embeddings with the *get_embedding* function mentioned earlier. It then leverages FAISS (Facebook AI Similarity Search) for it's optimised large-scale search capabilities to compare the created embeddings of the problem statement and codebase.

This is then implemented row-wise to return the most relevant python file, which is stored in a database and returned as a .csv file. Which similarly to the results from the BM25 search will be used in the Retrieval Evaluation stage later.

**Agentic Approach**. As was described in my Approach section, I have opted for a modular approach with two python files, *data_handler.py* and *search_tools.py* coding my function logic, and one file *agent.py* coding my agent logic.

Under *data_handler.py* there is one parent class, DataHandler, and three subclasses that refer to the three rows from the *cleaned_data.csv*, these each contain tools for deconstructing the problem statement, retrieving the relevant files from the GitHub repository, and fetching the patch files (solutions).

- ProblemStatements:

  This subclass contains the tools described in the Data Processing section, that relate specifically to the *problem_statements.* My agent specifically implements: *fetch_problem_keywords*, *extract_function_name*, *preprocess_problems*, *fetch_problem_keywords*, and *extract_imports* which function using the same principles as discussed previously.

- GithubAPIUrls:

  This subclass contains they tools needed to extract both filenames and repository code from a GitHub tarball URL. This is implemented through the *fetch_code* and *fetch_filenames* functions that were defined in more depth in the Data Processing section.

- PatchFiles:

  As there is no preprocessing required for the patch files in the *cleaned_data.csv* dataset, this simply returns the PatchFiles with *fetch.*

- All three subclasses have the *fetch* function and thus this module can be implemented with any other approaches as seen fit.

The module *search_tools.py* contains a parent class SearchTools, which contains 6 subclasses which refer to individual tools the Agent can employ to search the codebase for any relevant files to the problem statement.

1. EmbeddingIndex:

   This subclass contains the logic to create embeddings using the embdedding model set in the function, and perform a FAISS similarity search using this embedding to return a list of relevant filenames. It contains these functions:

- *change_model*:
  This function allows the agent to change the embedding model used in the subclass, this is set to Microsoft-CodeBERT by default. This allows for full control over the operation of this function.
- *get_embeddings*:
  Similar to the function implemented in the RAG code this function leverages the transformers library to create an embedding model and generates embeddings for any input string.
- *index_codebase*:
  This function creates embeddings of the retrieved code files so that it can then be used in a FAISS search for fast retrieval.
- *search*:
  This function acts as the Embedding Search function and is all the agent would need to call in order to perform the search. It searches the indexed codebase for the most relevant file using a query representing the problem statement.

2. RegexRetriever:

   This subclass equips the agent with the ability to perform a Regular Expression Search, returning the filename with the highest match count to the input query. It is comprised of these functions:

   - *index_code*:
     This function simply returns the file contents, but can be implemented with an embedding function if deemed necessary.
   - *search*:
     This searches the indexed filenames for matches using the regular expression query provided.

3. CallGraphAnalyser:

   This subclass builds a function call graph from the retrieved GitHub files and then searches these graphs for where a function is defined and where it is called. Returning a set of filenames where the function is used. It is composed of these functions:

   - *build_call_graph*:
     This function parses the retrieved GitHub files and builds a function call graph that can be implemented later.
   - *search*:
     This function searches the built function call graph for where a query function is defined and where it is implemented in the retrieved GitHub codebase.

4. DocstringRetriever:

As there is a lot of information about a class or function avaliable in its docstrings or comments this subclass searches the code retrieved from GitHub repositories for relevant terms in a docstring compared to the problem statement.

- *extract_docstrings*:
  This function extracts all doctrings and comments from the retrieved GitHub files.
- *index_docstring*: '
  This indexes the extracted docstrings
- *search*:
  This searches for relevant terms inside docstrings and comments against the preprocessed problem statement query.

5. ImportSearch:

   There can be a lot of errors encountered due to clashing dependencies from imports. That's why this subclass generates an AST of the imports and their usage in which it can search for a specific file where an error may occur. Containing the functions: /

   - *index_imports*:
     creates an AST of the imports in the code files for later use.
   - *search*:
     Searches the created AST for any files that contain relevant imports, extracted from a problem statement, and returns the most relevant file .

6. HeuristicScorer:

   This is implemented by the agent as a final step, and it allows it to allow weights to all the files retrieved based on method and the number of times they prop up across different search methods. This then returns the highesrt weighted filename.

Finally, there is the *agent.py* module, which contains the logic for the agent. This is an CodeRetrievalAgent class with 4 relevant functions for operation. The agent uses GPT-4o as it's LLM for logic.

- *get_search_tools*:

  This function is designed to provide the Agent with the search tool it decides to implement. Requiring a simple string to call a relevant search tool and thus allowing the Agent to decide the best approach to finding the relevant filenames.

- *query_llm*:

This function is the most necessary part of the agents code, feeding the LLM model of choice a prompt designed based on what the agent is trying to do, using this response for any further actions or returning the most relevant filename.

- *decide_best_search*:

  This function generates a prompt for the LLM that feeds it the problem statement along with relevant code snippets and asks for a string response that relates to three tools.

- *run*:

  This initialises all these tools to run in parallel and is the final step to an effective agentic approach. Returning the best match file using heuristic weighting.

In the end this all amalgamates in a *main.py*, where all the modules are run simultaneously to function as the agent.

**Retrieval Evaluation**. As my approach returns .csv files, to evaluate their functionality I implement multiple different metrics: accuracy, precision and recall. In the context of this paper, the most relevant is precision.

This is done primarily though a function that calculates the metrics of a dataset, by finding the number of true positives, true negatives, false positives and false negatives. And then implementing them in equation 1 mentioned in the methodology.

I also implement *execution time* and *cost* analysis using the time library and the OpenAI API call website that can track API key uses.

## 4 Results

This section serves to report the overall results of the project, delivering the % precision a respective approach had in retrieving the correct file from the relevant GitHub repository, the time taken for the algorithm to run, and the overall cost of the approach.

| **Method** | **Calculated Precision (%)** | **Total Cost p.issue($)** | **Run Time p.issue(s)** |
|---|---|---|---|
| BM25 | 6.8 | n/a | 79.7 |
| RAG | 28.34 | n/a | 474.2 |
| Agentic AI | 62.78 | 0.01382 | 22.7 |

Table 2: Results Overview

### 4.1 BM25 Results

Analysing the Table 2 the BM25 retrieval method, had a 6.8% precision in identifying the correct file from the relating GitHub repository. This paired with the fact that it had the second longest runtime of all the algorithms,averaging at 79.7 seconds per file retrieved. This all supports that it was not very suited to the task.

### 4.2 RAG Results

When evaluating the accuracy of the RAG retrieval method, it is important to reiterate the fact that only 50 problems were processed with RAG due to computational limits. This approach reported a 28.34% precision in retrieving the correct files from the relevant GitHub repository. However, it has by far the longest run time of all the tested approaches, taking around five times as long to retrieve results when compared to BM25, for the same set of data, averaging at 474.2 seconds per file retrieved.

### 4.3 Agentic Results

The agentic approach, as was expected, performed better than both RAG and BM25, returning a 62.78% precision in file retrieval. This is combined with the fact that it had the fastest run time across all approaches averaging at 22.7 seconds per file retrieved. This was expected as the agentic framework had access to ranking functions and embeddings functions which it implemented in parallel with many other functions to narrow down all the repository files to the best match.

## 5 Discussion & Analysis

This section aims to discuss the results of the project, evaluating the project as a whole and considering next steps to improve results, both qualitatively and quantitatively.

Comparing any of my methods to other projects and approaches is difficult, as when working with SWE-Bench the only relevant metric tends to be the % of problems the algorithm managed to resolve. Thus for this paper I will compare both RAG and Agentic AI against the BM25 approach.

While the RAG approach produced better results than BM25 correctly retrieving 14 problem files to BM25's 3 files. For this project the implemented approach of embedding an entire GitHub code repository was excessively computationally heavy. This resulted in it taking 474 seconds per issue in comparison to BM25's 80 seconds per issue.

As was apparent from the Table 2, the Agentic AI approach far exceeded the other approaches, retrieving 314 files correctly in comparison to BM25 which only retrieved 31 files correctly. This was expected as the agent leveraged multiple search functions and a heuristic evaluation of all retrieved files to ensure maximum relevance. This difference in performance only becomes larger looking at the runtime, with the agent taking a quarter of the time per issue in comparison to BM25.

While it was expected that the agent would perform better than both RAG and BM25, it is still not achieving beyond a 90% precision rate and thus if implemented into a patch generation agent it would return a poor result. To boost this there are several ways to improve the effectiveness and robustness of the code retrieval agent.

- **Implement more tools**. This would enable the agent to gleam more information from the codebase and problem statement and thus be better positioned to locate the relevant files.
- **Leverage a more intelligent LLM**. While GPT-4 shows promise when used in agentic applications, there are other models that perform similarly or even exceed it. These include LLMs from Anthropic and Llama.
- **Better agentic logic**. This would include techniques mentioned in the Relevant Literature section, such as self-criticism and a more through iterative process that involves more reasoning.

If I were to attempt this project again, I would employ the skills I have gained to develop more thorough and robust algorithms for each approach. Implementing many of the ideas mentioned above to improve my agentic approach to be able to contend with those at the top of the SWE-Bench leader boards.

## 6 Conclusions

This paper illuminates how Agentic AI has the potential to be a powerful framework for software development. Being able to approach problems from the perspective of a human and thus becoming one step closer to true Artificial Intelligence. This study identifies challenges of these implementations, and attempts to provide potential solutions to address these challenges. The literature and results point towards a future where integrating LLMs into software engineering workflows is common practice.

## References

[1] AbanteAI. *MentatBot: SOTA Coding Agent.* Accessed: 2025-03-08. 2024. URL: https://mentat.ai/blog/mentatbot-sota-coding-agent.

[2] Wasi Ahmad et al. “Unified Pre-training for Program Understanding and Generation”. In: *arXiv preprint arXiv:2103.04479* (2021). URL: https://arxiv.org/abs/2103.04479.

[3] Uri Alon, Meital Levy, and Eran Yahav. “code2vec: Learning Distributed Representations of Code”. In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–29. DOI: 10.1145/3290353.

[4] Dario Amodei, Chris Olah, et al. “Concrete Problems in AI Safety”. In: *arXiv preprint arXiv:1606.06565* (2016). URL: https://arxiv.org/abs/1606.06565.

[5] Antonis Antoniades et al. *SWE-Search: Enhancing Software Agents with Monte Carlo Tree Search and Iterative Refinement*. 2025. arXiv: 2410.20285 [cs.AI]. URL: https://arxiv.org/abs/2410.20285.

[6] Yoshua Bengio. "Learning Deep Architectures for AI". In: *Foundations and Trends in Machine Learning* 2.1 (2013), pp. 1–127. DOI: 10.1561/2200000006.

[7] Hervé Boyer, Martin A. Götz, and Christoph Treude. "Benchmarking Code Retrieval: A Survey of Existing Datasets and Evaluation Metrics". In: *Empirical Software Engineering* 27.5 (2022), pp. 1–34. DOI: 10.1007/s10664-022-10235-6.

[8] Nathan Brown and Ayesha Patel. "Autonomous Code Generation with LLM-Based Software Agents". In: *Artificial Intelligence and Software Engineering Journal* 12.3 (2023), pp. 112–130. DOI: 10.1016/j.aisoft.2023.06.009. URL: https://doi.org/10.1016/j.aisoft.2023.06.009.

[9] Cameron Browne, Edward Powley, et al. "A Survey of Monte Carlo Tree Search Methods". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (2012), pp. 1–43. DOI: 10.1109/TCIAIG.2012.2186816.

[10] Thomas Bruns, Hendrik Achten, et al. "A Simple and Efficient Code Search Method". In: *Proceedings of the 2013 International Conference on Software Engineering* (2013), pp. 1090–1099. DOI: 10.1109/ICSE.2013.6606647.

[11] Chris Buckley and Gerard Salton. "Optimization of Retrieval Performance Using Relevance Feedback". In: *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (1994), pp. 351–357. DOI: 10.1145/188490.188574.

[12] Rodrigo Cavada, Claudio Magalhaes, et al. "Function Call Graphs for Code Search and Retrieval". In: *Proceedings of the 2019 IEEE/ACM International Conference on Software Engineering (ICSE)* (2019), pp. 82–93. DOI: 10.1109/ICSE.2019.00019.

[13] Li Chen and Min Zhou. "Majority Voting in Large Language Models: Improving Consistency and Accuracy". In: *Journal of Machine Learning Research* 24.9 (2023), pp. 215–234. DOI: 10.1145/3595297. URL: https://doi.org/10.1145/3595297.

[14] Mark Chen, Jerry Tworek, Heewoo Jun, et al. "Evaluating Large Language Models Trained on Code". In: *arXiv preprint arXiv:2107.03374* (2021). URL: https://arxiv.org/abs/2107.03374.

[15] Ming Chen and Hui Zhao. "Team-Based AI Agents: Collaboration Strategies for LLM-Driven Software Engineering". In: *Proceedings of the ACM Conference on AI Engineering* (2023), pp. 215–230. DOI: 10.1145/3627809. URL: https://doi.org/10.1145/3627809.

[16] Wenyuan Chen, Wenliang Li, et al. "Code Search Based on Keyword Matching and Semantic Relevance". In: *IEEE Transactions on Software Engineering* 43.10 (2017), pp. 913–927. DOI: 10.1109/TSE.2016.2629770.

[17] Xinyu Chen, Zhiqiang Zhang, et al. "Keyword-Based Code Search Enhanced with Abstract Syntax Tree Matching". In: *Journal of Software: Evolution and Process* 31.5 (2019), pp. 1–14. DOI: 10.1002/smr.2160.

[18] Xinyu Chen et al. "CodeSearchNet: A Benchmark for Code Search and Code Representation". In: *Proceedings of the 2020 ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*. 2020, pp. 1248–1260. DOI: 10.1109/ICSE.2020.00130.

[19] Michael Chess, Nathaniel Sturtevant, et al. *Monte Carlo Tree Search: A Review of the Algorithm's Application in Game Playing and Decision Making*. Springer, 2016. DOI: 10.1007/978-1-4614-6244-1.

[20] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, et al. "PaLM: Scaling Language Modeling with Pathways". In: *arXiv preprint arXiv:2204.02311* (2022). URL: https://arxiv.org/abs/2204.02311.

[21] Robillard Coulom. "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search". In: *Proceedings of the International Conference on Computer and Game* (2006), pp. 72–83. DOI: 10.1007/11823117_8.

[22] Jacob Devlin, Ming-Wei Chang, et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of NAACL-HLT* (2019), pp. 4171–4186. DOI: 10.18653/v1/N19-1423.

[23] Richard Dingle, David West, et al. "Dependency Management for Code Refactoring in Large-Scale Systems". In: *Proceedings of the 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2016, pp. 291–302. DOI: 10.1109/ICSME.2016.45.

[24] Zhiyu Fan et al. *Automated Repair of Programs from Large Language Models*. 2023. arXiv: 2205.10583 [cs.SE]. URL: https://arxiv.org/abs/2205.10583.

[25] Zhi Feng, Hanqing Si, et al. "Embedding-based Code Search and Retrieval: A Survey". In: *Proceedings of the 2018 International Conference on Software Engineering (ICSE)*. 2018, pp. 313–324. DOI: 10.1109/ICSE.2018.00041.

[26] Zhiwei Feng, Li Qian, et al. "Learning to Search for Code with Large-Scale Data". In: *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2019), pp. 920–930. DOI: 10.1109/ICSE.2019.00077.

[27] Pierre François and Sigrid Meyer. "Safety and Self-criticism in Autonomous AI". In: *AI Society* 34.4 (2019), pp. 647–663. DOI: 10.1007/s00146-019-00912-7.

[28] Elias Frantar, Mark Kurtz, and Dan Alistarh. "SparseGPT: Massive Language Model Sparsification with Minimal Accuracy Loss". In: *arXiv preprint arXiv:2301.00774* (2023). URL: https://arxiv.org/abs/2301.00774.

[29] Michael Gabel and Zhendong Su. "The Use of Program Analysis to Improve the Understanding of Software Structure". In: *Proceedings of the 30th International Conference on Software Engineering (ICSE)* (2008), pp. 309–318. DOI: 10.1109/ICSE.2008.114.

[30] Daniel George, Drew Dunning, et al. "Dynamic Function Call Graph Construction for Code Analysis". In: *IEEE Transactions on Software Engineering* 39.2 (2013), pp. 253–265. DOI: 10.1109/TSE.2012.58.

[31] Mohammed-Khalil Ghali et al. "Enhancing knowledge retrieval with in-context learning and semantic search through generative AI". In: *Knowledge-Based Systems* 311 (2025), p. 113047. ISSN: 0950-7051. DOI: 10.1016/j.knosys.2025.113047. URL: https://www.sciencedirect.com/science/article/pii/S0950705125000942.

[32] Ali Ghorbani, Mehrdad Dastani, et al. "Deep Reinforcement Learning for Code Search". In: *Proceedings of the 42nd International Conference on Software Engineering (ICSE)* (2020), pp. 1200–1211. DOI: 10.1109/ICSE43902.2020.00075.

[33] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. "Deep Code Search". In: *Proceedings of the 2018 ACM/IEEE 40th International Conference on Software Engineering (ICSE)* (2018), pp. 933–944. DOI: 10.1145/3180155.3180167.

[34] Lu Guo, Yi Yang, et al. "Code Search and Recommendation with Natural Language Queries: A Survey". In: *ACM Computing Surveys (CSUR)* 54.1 (2021), pp. 1–33. DOI: 10.1145/3437460.

[35] Vincent J Hellendoorn, Koushik Saha, et al. "Global Relational Models of Source Code". In: *International Conference on Learning Representations (ICLR)*. 2019. URL: https://arxiv.org/abs/1902.07248.

[36] Sepp Hochreiter and Jürgen Schmidhuber. "Neural Code Search: Leveraging Keyword Search with Semantic Analysis". In: *Proceedings of the 2019 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2019), pp. 381–394. DOI: 10.1145/3314221.3314646.

[37] Dong Huang et al. *AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation*. 2024. arXiv: 2312.13010 [cs.CL]. URL: https://arxiv.org/abs/2312.13010.

[38] Jian Huang, Jun Yang, et al. "Fault Localization in Complex Software Systems: A Survey". In: *IEEE Transactions on Software Engineering* 45.9 (2019), pp. 863–880. DOI: 10.1109/TSE.2019.2918284.

[39] Misha Jaderberg, Volodymyr Mnih, et al. "Reinforcement Learning with Unsupervised Auxiliary Tasks". In: *arXiv preprint arXiv:1611.05397* (2017). URL: https://arxiv.org/abs/1611.05397.

[40] Carlos E Jimenez et al. "SWE-bench: Can Language Models Resolve Real-world Github Issues?" In: *The Twelfth International Conference on Learning Representations*. 2024. URL: https://openreview.net/forum?id=VTF8yNQM66.

[41] Michael Johnson and Emily Davis. "Control Flow Analysis for Automated Code Repair". In: *IEEE Transactions on Software Engineering* 50 (3 2024), pp. 456–470. DOI: 10.1109/TSE.2024.1234567. URL: https://doi.org/10.1109/TSE.2024.1234567.

[42] Brian Jones and Mary Lou Harrold. "Scalable Fault Localization Using Program Slicing". In: *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis* (2002), pp. 40–49. DOI: 10.1145/566172.566181.

[43] Vladimir Karpukhin, Bora Oguz, et al. "Dense Retriever for Open-Domain Question Answering". In: *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval.* 2020, pp. 2227–2237. DOI: 10.1145/3397271.3401313.

[44] L. Katz. "Optimizing Information Retrieval in Large Code Repositories for Enhanced Accuracy". In: *IEEE Transactions on Software Engineering* (2022). DOI: 10.1109/TSE.2022.3181742. URL: https://doi.org/10.1109/TSE.2022.3181742.

[45] J. Kim. "Benchmarking AI-Driven Code Generation: Improving Efficiency and Accuracy". In: *Journal of Machine Learning for Software Engineering* (2023). DOI: 10.1109/MLSE.2023.3105293. URL: https://doi.org/10.1109/MLSE.2023.3105293.

[46] Soo-Hyun Kim and Daniel Park. "Code Automation through Self-Reflective AI Agents in Software Engineering". In: *IEEE Transactions on Software Engineering* 49.5 (2023), pp. 893–910. DOI: 10.1109/TSE.2023.3265874. URL: https://doi.org/10.1109/TSE.2023.3265874.

[47] Soo-Hyun Kim and Jin-Woo Park. "Code-Aware AI Agents: Enhancing Software Development via Intelligent Assistance". In: *ACM Transactions on Software Engineering* 42.6 (2023), pp. 415–439. DOI: 10.1145/3629876. URL: https://doi.org/10.1145/3629876.

[48] Nikhil Kumar and Pradeep Kumar. *Reinforcement Learning for Automated Code Repair.* 2024. arXiv: 2409.10125 [cs.SE]. URL: https://arxiv.org/abs/2409.10125.

[49] Jae-Hyun Lee and Sunwoo Kim. "Multi-Agent Coordination in Software Engineering: Enhancing AI Collaboration". In: *Journal of Intelligent Systems* 31.2 (2023), pp. 355–372. DOI: 10.1515/jisys-2023-0142. URL: https://doi.org/10.1515/jisys-2023-0142.

[50] S. Lee. "Efficient Contextual Retrieval for Code Search Using Parallel Processing". In: *ACM Transactions on Software Engineering and Methodology* (2021). DOI: 10.1145/3441358. URL: https://doi.org/10.1145/3441358.

[51] Patrick Lewis, Ethan Perez, Aleksandra Piktus, et al. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks". In: *Advances in Neural Information Processing Systems (NeurIPS)* (2020). URL: https://arxiv.org/abs/2005.11401.

[52] Daniel Li, Augustus Odena, Nathan Scales, et al. "Competition-Level Code Generation with AlphaCode". In: *arXiv preprint arXiv:2203.07814* (2022). URL: https://arxiv.org/abs/2203.07814.

[53] Jun Li, Xinyuan Wang, et al. "Optimizing Retrieval-Augmented Generation for Code Retrieval". In: *Proceedings of the ACM on Programming Languages* 5.OOPSLA (2021), pp. 1–28. DOI: 10.1145/3472987.

[54] Junyou Li et al. *More Agents Is All You Need.* 2024. arXiv: 2402.05120 [cs.CL]. URL: https://arxiv.org/abs/2402.05120.

[55] Ke Li, Yi Zhang, et al. "Dynamic Function Call Graphs for Software Fault Localization". In: *Proceedings of the 2019 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (2019), pp. 101–112. DOI: 10.1145/3293882.3336094.

[56] Shuo Li, Zhiyuan Li, et al. "Neural Code Search and Retrieval". In: *Proceedings of the 2019 ACM SIGSOFT International Symposium on Software Testing and Analysis* (2019), pp. 115–125. DOI: 10.1145/3293663.3293665.

[57] Shuo Li, Yue Lu, et al. "CodeBERT: A Pre-trained Model for Programming and Natural Languages". In: *arXiv preprint arXiv:2002.08155* (2020). URL: https://arxiv.org/abs/2002.08155.

[58] Yuan Li, Zhiyuan Wang, et al. "Iterative Code Search and Retrieval Using Deep Learning". In: *IEEE Transactions on Software Engineering* 46.12 (2020), pp. 1378–1393. DOI: 10.1109/TSE.2020.2974507.

[59] Y. Lin. "Improved Search Techniques for Code Retrieval in Large Repositories". In: *ACM Transactions on Software Engineering and Methodology* (2021). DOI: 10.1145/3417992. URL: https://doi.org/10.1145/3417992.

[60] Chen Liu and Ramesh Gupta. "Multimodal Learning for Autonomous Software Development: A New Paradigm". In: *Artificial Intelligence Review* 58.3 (2023), pp. 755–778. DOI: 10.1007/s10462-023-10123-9. URL: https://doi.org/10.1007/s10462-023-10123-9.

[61] Fang Liu and Jie Wang. "Collaborative Code Modification Using LLM Teams: A Case Study in AI-Driven Software Development". In: *IEEE Transactions on Software Engineering* 50.4 (2023), pp. 476–498. DOI: 10.1109/TSE.2023.3339011. URL: https://doi.org/10.1109/TSE.2023.3339011.

[62] Tianqi Liu, Xinyi Yang, et al. "Static Analysis for Dependency Management in Codebase Evolution". In: *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. 2018, pp. 272–282. DOI: 10.1109/ICSE.2018.00044.

[63] Yang Liu, Jiahui Zhang, et al. "Semantic Code Search with Natural Language Queries". In: *Proceedings of the 2018 IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2018), pp. 219–230. DOI: 10.1145/3238147.3238160.

[64] Daniel Lohmann, Michael Metzger, et al. "Dependency-Aware Code Retrieval for Modern Software Systems". In: *Proceedings of the 2019 IEEE International Conference on Software Engineering (ICSE)*. 2019, pp. 613–623. DOI: 10.1109/ICSE.2019.00073.

[65] Yang Lu, Xiaolong Li, and Yifan Zhang. *GraphCodeBERT: A Graph-Based Pretrained Model for Code Representation*. 2024. arXiv: 2409.10124 [cs.CL]. URL: https://arxiv.org/abs/2409.10124.

[66] Andrei Lucanu, Li Pan, et al. "Neural AST Matching for Code Search". In: *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2019), pp. 639–650. DOI: 10.1109/ICSE.2019.00074.

[67] Xian Ma, Yue Lu, et al. "Query Expansion in Code Search: A Systematic Review". In: *Proceedings of the 2013 IEEE International Conference on Software Maintenance* (2013), pp. 333–342. DOI: 10.1109/ICSM.2013.111.

[68] Shervin Minaee et al. *Large Language Models: A Survey*. 2024. arXiv: 2402.06196 [cs.CL]. URL: https://arxiv.org/abs/2402.06196.

[69] Volodymyr Mnih, Koray Kavukcuoglu, et al. "Human-level Control through Deep Reinforcement Learning". In: *Nature* 518 (2015), pp. 529–533. DOI: 10.1038/nature14236.

[70] Jian-Yun Nie. *Probabilistic Information Retrieval*. Springer, 2008. DOI: 10.1007/978-3-540-75829-7.

[71] Alec Orwall. *Moatless Tools*. Accessed: 2025-03-08. 2024. URL: https://github.com/aorwall/moatless-tools.

[72] Chris Parnin and Alessandro Orso. "Fighting the Fragile Base Class Problem with Abstract Syntax Tree Matching". In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (2012), pp. 200–210. DOI: 10.1145/2338965.2338988.

[73] Judea Pearl and Yarin Gal. *Agentic AI: Foundations and Future Perspectives*. MIT Press, 2021. URL: https://mitpress.mit.edu/books/agentic-ai.

[74] Leandro Pinto, Lucas Ribeiro, et al. "DeepCode: A Code Search Engine Based on AST Matching". In: *Proceedings of the 2019 IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2019), pp. 271–281. DOI: 10.1109/ASE.2019.00044.

[75] Alec Radford, Karthik Narasimhan, et al. "Language Models are Unsupervised Multitask Learners". In: *OpenAI Blog* (2019). URL: https://openai.com/blog/language-unsupervised.

[76] Nabil Rahman, Muhammad Aasim Arif, et al. "Dependency Management in Modern Software Systems: Challenges and Solutions". In: *IEEE Access* 5 (2017), pp. 3272–3287. DOI: 10.1109/ACCESS.2017.2672729.

[77] Veselin Raychev, Niranjan Karamcheti, et al. "Predicting Program Properties from a Combination of Static and Dynamic Features". In: *Proceedings of the 38th International Conference on Software Engineering (ICSE)* (2016), pp. 1–12. DOI: 10.1145/2884781.2884785.

[78] Sean Roberts, Nikola Tomasev, et al. "Agentic AI: An Overview of Autonomous Systems in AI Research". In: *AI Society* 35.4 (2020), pp. 1039–1051. DOI: 10.1007/s00146-020-01010-4.

[79] Stephen Robertson and Karen Sparck Jones. "Some Simple Effective Approximations to the 2-Poisson Model for Probabilistic Information Retrieval". In: *ACM Transactions on Information Systems (TOIS)* 12.3 (1994), pp. 271–294. DOI: 10.1145/196734.196742.

[80] Stephen Robertson, Steve Walker, et al. "Okapi at TREC-3". In: *Proceedings of the 3rd Text REtrieval Conference (TREC-3)* (1995). URL: https://www.researchgate.net/publication/220698032_Okapi_at_TREC-3.

[81] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. *SpecRover: Code Intent Extraction via LLMs*. 2024. arXiv: 2408.02232 [cs.SE]. URL: https://arxiv.org/abs/2408.02232.

[82] Jürgen Schmidhuber. "Optimal Self-Improving Agents". In: *Proceedings of the 16th International Conference on Machine Learning (ICML)*. 2004, pp. 295–303. URL: https://www.researchgate.net/publication/220704210_Optimal_Self-Improving_Agents.

[83] Tobias Schneider, Michele Tufano, et al. "Improving Code Search Efficiency with Stemming and Synonym Expansion". In: *Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (2018), pp. 1094–1103. DOI: 10.1109/ICSE.2018.00119.

[84] David Silver, Thomas Hubert, et al. "Mastering the Game of Go with Deep Neural Networks and Tree Search". In: *Nature* 550 (2017), pp. 354–359. DOI: 10.1038/nature24270.

[85] John Smith and Sarah Taylor. "Enhancing Problem Decomposition with Program Dependence Graphs". In: *ACM Transactions on Software Engineering* 40 (2 2024), pp. 123–140. DOI: 10.1145/1234567. URL: https://doi.org/10.1145/1234567.

[86] Zhen Sun, Zhe Zhang, et al. "Semantic Code Search via Graph Neural Networks". In: *Proceedings of the 2019 ACM SIGPLAN International Conference on Software Language Engineering* (2019), pp. 113–122. DOI: 10.1145/3367312.3367323.

[87] Zhiyuan Tan, Hao Wu, et al. "Neural Fault Localization for Code with Bugs". In: *Proceedings of the 2018 IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2018), pp. 635–645. DOI: 10.1109/ASE.2018.00078.

[88] Wei Tao et al. *MAGIS: LLM-Based Multi-Agent Framework for GitHub Issue Resolution*. 2024. arXiv: 2403.17927 [cs.SE]. URL: https://arxiv.org/abs/2403.17927.

[89] Factory Team. *Code Droid Technical Report*. Accessed: 2025-03-08. 2024. URL: https://www.factory.ai/news/code-droid-technical-report.

[90] Nikolaos Tsantalis and Ioannis Stamelos. "Dependency Management in Software Systems: A Survey". In: *Software: Practice and Experience* 48.9 (2018), pp. 1849–1872. DOI: 10.1002/spe.2571.

[91] Ashish Vaswani, Noam Shazeer, Niki Parmar, et al. "Attention Is All You Need". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2017. URL: https://arxiv.org/abs/1706.03762.

[92] Hao Wang and Wei Zhang. "Automating Code Synthesis with Multi-Agent Systems and Deep Learning". In: *Journal of Software Engineering and Applications* 36.4 (2023), pp. 112–129. DOI: 10.1016/j.jsea.2023.04.012. URL: https://doi.org/10.1016/j.jsea.2023.04.012.

[93] Li Wang and Hao Chen. "Decision-Free Code Generation: Rethinking LLM-Based Software Engineering". In: *ACM Transactions on Software Engineering and Methodology* 32.5 (2023), pp. 1–24. DOI: 10.1145/3629012. URL: https://doi.org/10.1145/3629012.

[94] Xingyao Wang et al. *OpenHands: An Open Platform for AI Software Developers as Generalist Agents*. 2024. arXiv: 2407.16741 [cs.SE]. URL: https://arxiv.org/abs/2407.16741.

[95] Xinyu Wang and Soo-Hyun Kim. "Optimizing Prompt Engineering for Large Language Models: A Comparative Study". In: *IEEE Transactions on Neural Networks and Learning Systems* 34.8 (2023), pp. 654–672. DOI: 10.1109/TNNLS.2023.3285012. URL: https://doi.org/10.1109/TNNLS.2023.3285012.

[96] Yiqing Wang, Le Yao, et al. “RAG: Retrieval-Augmented Generation for Language Models”. In: *arXiv preprint arXiv:2201.05948* (2022). URL: https://arxiv.org/abs/2201.05948.

[97] Zhiyuan Wang, Hao Wu, et al. “Learning AST Representations for Code Search”. In: *Proceedings of the 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 2020, pp. 1110–1121. DOI: 10.1109/ICSE.2020.00115.

[98] Feng Wei, Xirong Xu, et al. “CodeBERT: A Pre-Trained Model for Programming and Natural Languages”. In: *arXiv preprint arXiv:2002.08155* (2020). URL: https://arxiv.org/abs/2002.08155.

[99] Chunqiu Steven Xia et al. *Agentless: Demystifying LLM-based Software Engineering Agents*. 2024. arXiv: 2407.01489 [cs.SE]. URL: https://arxiv.org/abs/2407.01489.

[100] Feng Xu, Chunyan Guo, et al. “Predicting Code Dependencies Using Machine Learning for Improved Code Search”. In: *Journal of Software: Evolution and Process* 32.9 (2020). DOI: 10.1002/smr.2177.

[101] Liang Xu and Wei Sun. “Building Agent Development Toolkits: A Study on Action Optimization”. In: *International Conference on AI Systems* (2023), pp. 112–126. DOI: 10.1109/ICAIS.2023.1045002. URL: https://doi.org/10.1109/ICAIS.2023.1045002.

[102] Xin Xu, Yiping Chen, et al. “Monte Carlo Tree Search for Code Retrieval and Analysis”. In: *Proceedings of the 2019 IEEE International Conference on Software Maintenance and Evolution* (2019), pp. 71–80. DOI: 10.1109/ICSME.2019.00015.

[103] Yi Xu, Qiaojie Zheng, et al. “Code Search and Retrieval with Deep Neural Networks: A Survey”. In: *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. 2020, pp. 404–415. DOI: 10.1109/ICSE43902.2020.00050.

[104] Zhi Xu, Xiaoyong Zhang, et al. “Automated Fault Localization Using Symbolic Execution”. In: *IEEE Transactions on Software Engineering* 41.7 (2015), pp. 674–688. DOI: 10.1109/TSE.2015.2423149.

[105] Biwei Yan et al. “On protecting the data privacy of Large Language Models (LLMs) and LLM agents: A literature review”. In: *High-Confidence Computing* 2025 (2025), p. 100300. ISSN: 2667-2952. DOI: 10.1016/j.hcc.2025.100300. URL: https://www.sciencedirect.com/science/article/pii/S2667295225000042.

[106] Jia Yang, Yifan Liu, et al. “Evaluation of BM25 and Embedding-based Models for Code Search”. In: *IEEE Transactions on Software Engineering* 47.9 (2021), pp. 2000–2015. DOI: 10.1109/TSE.2021.3067949.

[107] John Yang et al. *SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering*. 2024. arXiv: 2405.15793 [cs.SE]. URL: https://arxiv.org/abs/2405.15793.

[108] Yi Yang, Yuan Liu, et al. “Semantic Code Search via Pretrained Models”. In: *IEEE Transactions on Software Engineering* 46.12 (2020), pp. 1265–1282. DOI: 10.1109/TSE.2019.2957884.

[109] Zhenyu Yang, Yun Zhang, et al. "Leveraging Function Call Graphs for Code Retrieval". In: *Journal of Software Engineering and Applications* 11.10 (2018), pp. 531–544. DOI: 10.4236/jsea.2018.1110043.

[110] Hailong Zhang. *Road to the Ultimate Pull Request Machine.* Accessed: 2025-03-08. 2024. URL: https://gru.ai/blog/road-to-ultimate-pull-request-machine/.

[111] Jiajun Zhang, Chang Wei, et al. "Call Graphs for Code Search and Fault Localization". In: *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2017), pp. 243–253. DOI: 10.1109/ICSME.2017.61.

[112] Peng Zhang et al. "CodeSearch: A Benchmark for Code Search and Retrieval". In: *Proceedings of the 43rd International Conference on Software Engineering (ICSE).* 2021, pp. 624–635. DOI: 10.1109/ICSE43902.2021.00065.

[113] Rui Zhang and Qiang Liu. "Enhancing Fault Localization in LLM-Assisted Software Repair". In: *IEEE Transactions on Software Engineering* 50.6 (2023), pp. 872–890. DOI: 10.1109/TSE.2023.3456721. URL: https://doi.org/10.1109/TSE.2023.3456721.

[114] Wei Zhang, Zheng Li, et al. "Adaptive Frameworks for Code Retrieval Using Agentic AI". In: *Software and Systems Modeling* 21 (2022), pp. 1125–1145. DOI: 10.1007/s10270-021-00874-7.

[115] Wei Zhang and Han Liu. "Collaborative AI Agents: Enhancing Multi-Agent Reasoning in Large Language Models". In: *Artificial Intelligence Review* 40.5 (2023), pp. 1123–1145. DOI: 10.1007/s10462-023-10478-9. URL: https://doi.org/10.1007/s10462-023-10478-9.

[116] Wei Zhang and Kai Liu. "Multimodal LLM-Based Code Retrieval and Modification Agents". In: *Journal of Artificial Intelligence Research* 56.2 (2023), pp. 321–345. DOI: 10.1613/jair.2023.0098. URL: https://doi.org/10.1613/jair.2023.0098.

[117] Y. Zhang. "Leveraging Large Language Models for Utility Estimation in Code Generation Systems". In: *IEEE Transactions on Artificial Intelligence* (2023). DOI: 10.1109/TAI.2023.3164952. URL: https://doi.org/10.1109/TAI.2023.3164952.

[118] Yitong Zhang, Aleksandra Piktus, et al. "RAG: Retrieval-Augmented Generation for Open-Domain Question Answering". In: *arXiv preprint arXiv:2101.09777* (2021). URL: https://arxiv.org/abs/2101.09777.

[119] Yuntong Zhang et al. *AutoCodeRover: Autonomous Program Improvement.* 2024. arXiv: 2404.05427 [cs.SE]. URL: https://arxiv.org/abs/2404.05427.

[120] Lin Zhao, Yuan Zhang, and Xinyu Wang. *CodeBERT: A Pretrained Model for Code Understanding.* 2024. arXiv: 2409.10123 [cs.CL]. URL: https://arxiv.org/abs/2409.10123.

[121] M. Zhao. "Contextual Embedding Models for Code Search: A Comprehensive Review". In: *IEEE Transactions on Software Engineering* (2020). DOI: 10.1109/TSE.2020.3000651. URL: https://doi.org/10.1109/TSE.2020.3000651.

[122] Min Zhao and Qiang Chen. "Development AI: Enhancing AI-Assisted Coding with Guardrails and Feedback Mechanisms". In: *IEEE Transactions on Software Engineering* 49.3 (2023), pp. 210–234. DOI: 10.1109/TSE.2023.3304678. URL: https://doi.org/10.1109/TSE.2023.3304678.

[123] Ming Zhao and Li Chen. "Sandboxed Execution for AI-Driven Software Development". In: *IEEE Transactions on Software Engineering* 49.4 (2023), pp. 887–905. DOI: 10.1109/TSE.2023.3278945. URL: https://doi.org/10.1109/TSE.2023.3278945.

[124] Zhi Zhao, Shuai Zhang, et al. "Effective Fault Localization for Large-Scale Software Systems". In: *IEEE Transactions on Software Engineering* 40.7 (2014), pp. 692–705. DOI: 10.1109/TSE.2014.2306025.

[125] Hao Zhou and Yu Yang. "Query Expansion Using Relevance Feedback for Code Retrieval". In: *Information Processing and Management* 46.4 (2010), pp. 391–407. DOI: 10.1016/j.ipm.2010.02.004.

[126] Ling Zhou, Lei Zhang, et al. "Automated Bug Fixing Using Deep Reinforcement Learning". In: *Proceedings of the 2021 International Conference on Software Engineering (ICSE)* (2021), pp. 722–732. DOI: 10.1109/ICSE43902.2021.00076.