CoDescribe: An Intelligent Code Analyst for Enhancing Productivity and Software Quality

Souvick Das, Novarun Deb, Agostino Cortesi, and Nabendu Chaki

Abstract Automating the documentation, explanation, and modification of source code has been an area of intense research interest for several decades. This paper proposes a novel framework for analyzing source codes to generate natural language summaries, answer complex user queries by reasoning, and even support syntactic error checking and code conversion to other languages. The framework incorporates transformer-based models and knowledge graphs to understand the context of the code and the user's queries. The contributions of this proposed approach represent a significant leap forward by overcoming the limitations of the most advanced Chat-GPT model. The framework ensures the accuracy and reliability of the responses while handling multiple source code files efficiently, making it a valuable asset for software developers and stakeholders.

Keywords Code explainability · Semantic search · Knowledge graph · Question-answering

1 Introduction

In the realm of software engineering, source code summarization and explainability of source code have been critical areas of interest for decades. The purpose of

S. Das (✉) · A. Cortesi
Department of Environmental Science, Informatics, and Statistics, Ca' Foscari University, Venezia 30172, Italy
e-mail: souvick.das@unive.it

A. Cortesi
e-mail: cortesi@unive.it

N. Deb
Indian Institute of Information Technology, Vadodara, Gandhinagar, Gujrat 382028, India
e-mail: novarun_deb@iiitvadodara.ac.in

N. Chaki
Department of Computer Science and Engineering, University of Calcutta, 67 Sukanta Park, Kolkata 700101, India
e-mail: nabendu@ieee.org

R. Chaki et al. (eds.), *Applied Computing for Software and Smart Systems*, Lecture Notes in Networks and Systems 781, https://doi.org/10.1007/978-981-99-7783-3_11

writing source code summaries is to provide natural language descriptions of subroutines, aiding programmers in quickly grasping their purpose without sifting through the underlying source code. However, this task has proven to be quite challenging, as programmers often avoid writing such proper comments in the source codes, documentation, and summaries of codes. Despite its importance, source code explanation is often neglected during software development due to time constraints or a lack of awareness of its benefits. However, the consequences of not having well-documented source code can be significant, leading to increased debugging time, difficulty in maintaining and updating the code, and even the complete abandonment of the codebase. This has led to a high demand for automated solutions, which has encouraged intense research interest in this area.

Over the last decades, deep learning-based source code summarization approaches have emerged as the predominant method for automating the process. These methods leverage massive datasets of code and summaries to train attention-based encoder–decoder models. While these models have mostly replaced previous approaches such as sentence templates or IR-based keyword extraction. Currently, there are several deep learning-based approaches to analyze source code, each focusing on solving larger problems such as defect prediction [6], refactoring source codes [12], and code explanation and summarization [4, 5, 13]. There are various techniques [5, 6] to analyze code, such as treating it as text, using contextual knowledge from other code, modeling its structure with RNNs or GNNs, and applying encoder–decoder models, followed by generating a summary. Recent research has focused on developing more sophisticated representations of source code to improve prediction accuracy [5, 10]. These approaches often require extensive training and testing of deep learning models on appropriate datasets. However, current state-of-the-art lacks such approaches that not only explain code and generate documentation for the source code but also answer complex user queries by reasoning.

In this paper, we propose a novel framework, named CoDescribe, for source code summarization and explanation that not only generates natural language summaries but also provides answers to complex user queries by reasoning. Our framework leverages the latest advancements in natural language processing and deep learning to develop a model that can understand the context of the code and the user's queries. Specifically, we utilize a combination of transformer-based models to capture the structural and semantic information of the code. On the other hand, knowledge graphs have been used to represent the relationships between different components and concepts of source codes. By incorporating these graphs, we can enable our model to reason over the code and the user queries. Overall, our proposed approach represents a significant leap forward in automating the documentation, explanation, and modification of source code. Since this approach has the potential to greatly improve the maintainability and explainability of software systems, it reduces the debugging time and effort for the developers and making it valuable asset not only for individual stakeholders, but also for the entire development team.

The main functionalities of CoDescribe can be summarized as follows:

1. A detailed and abstract code explanation.

2. Customized documentation generation of the given source code.
3. Syntactic error checking and alteration of code detection.
4. Conversion of code into other programming languages.
5. Chat-based conversation regarding the provided source codes.

As will be discussed in Sect. 7, the main novelty of this approach with respect to existing tools supporting software documentation and error checking is its ability to handle multiple source code files and to capture their interconnectivity.

The rest of the paper is structured as follows:

Section 2 provides background concepts such as pre-trained language models, knowledge graph, and semantic search. We present our proposed framework in Sect. 3. In Sect. 4, we elaborate on the experimental evaluation of the framework. We present the detailed analysis of the results in Sect. 5. Section 6 highlights some threats to the validity of the framework. In Sect. 7, we present the existing state of the art for source code analysis toward summarization and code explainability. We also present the comparative analysis with the proposed methods and current state of the art. Finally, Sect. 8 concludes the paper by summarizing our contributions and by highlighting some future directions.

2 Preliminaries

This section of the research paper serves as an introduction to the key concepts necessary for a comprehensive understanding of the subsequent sections. In this section, we provide an overview of the foundational concepts of pre-trained language models, semantic search mechanisms, and how semantic search can be enhanced using a knowledge graph. By familiarizing the reader with these essential concepts, we aim to provide a solid foundation for the main framework of the research.

2.1 Pre-trained Language Models

Pre-trained language models (PLMs) are a type of deep learning model that have been trained on large amounts of text data and can be fine-tuned for various natural language processing tasks. These models have been shown to achieve state-of-the-art results on a wide range of NLP benchmarks, and their use has become widespread in the field of NLP. In recent years, there has been an increasing interest in developing pre-trained language models specifically for code, and several models have emerged that can be used for source code summarization and explainability. In this section, we provide a brief overview of some of the most prominent pre-trained language models that have been developed for code.

CodeT5 Model CodeT5 [26] is a variant of the T5 [22] language model, specifically trained for source code. It is based on the Transformer architecture and is trained on a large-scale dataset of code snippets and natural language descriptions. More specifically, it is trained on a massive dataset of over 750,000 methods and their corresponding documentation from GitHub, as well as a large number of methods and descriptions from Stack Overflow. CodeT5 has been shown to outperform previous state-of-the-art models in various code-related tasks, such as code completion and code summarization [26].

GPT-3.5 One of the most well-known PLMs is the Generative Pre-trained Transformer 3 (GPT-3) model [2], developed by OpenAI. GPT-3 is a transformer-based model that has been trained on a massive dataset of over 45TB of text data, and has over 175 billion parameters. The model can perform a wide range of NLP tasks, including language translation, question-answering, summarization, and text generation.

Recently, a new version of GPT-3, called GPT-3.5 [19], has been released. GPT-3.5 is an improved version of GPT-3. GPT-3.5 is a much larger model than GPT-3.0, with 355 billion parameters compared to 175 billion. This increased capacity allows for more accurate and natural-sounding responses, making it more useful for tasks like language translation and text summarization. The chat variant model of the GPT-3.5, called ChatGPT, is a language model that can generate text in various styles and for different purposes with high precision, detail, and coherence. The model was fine-tuned using a combination of supervised and reinforcement learning techniques, with a focus on reinforcement learning from human feedback (RLHF) to minimize negative outputs. We have chosen GPT-3.5 as the foundational model for our framework, as it enables us to respond to user queries on source code in a chat-based environment.

LLaMA The LLaMA [25] model is a collection of foundation language models ranging from 7B to 65B parameters. It was introduced by Meta AI as a state-of-the-art foundational large language model designed to help researchers advance their work in this subfield of AI. It is an auto-regressive language model based on the transformer architecture. The models were trained on trillions of tokens using publicly available datasets exclusively. LLaMA-13B outperforms GPT-3 on most benchmarks despite being 10% smaller. Given its generative nature, the LLaMA model is inherently capable of generating text. To harness this ability, we fine-tune the model to produce natural language descriptions of source code.

2.2 Semantic Search

Semantic search is a type of search technology that uses natural language processing (NLP) algorithms to understand the context and meaning of search queries [7]. Unlike traditional keyword-based search, which matches exact words or phrases in a document, semantic search aims to understand the intent behind the query and provide results that are conceptually related to the user's search terms.

One of the most commonly used methods for semantic search is the use of vector spaces, where documents are represented in a semantic vector space that captures the knowledge contained within the text. When a user makes a search query, semantic search maps the query to the same vector space, allowing it to retrieve documents that contain semantically related content.

Unlike traditional keyword-based search methods, which rely on exact word matches, the semantic search focuses on the actual meaning of documents. For example, a user searching for "Model Driven Engineering" would also retrieve documents containing information on "MDE," as the semantic model recognizes that these terms are semantically related.

Semantic search is a search methodology that gives greater consideration to the user's intent and identifies the most appropriate documents that match that intent. Semantic search is not just about finding documents with matching keywords or phrases, but about finding results that are truly helpful to the user. By using natural language processing and machine learning techniques, semantic search can provide much better results and speed up the information discovery process for users. There are two types of semantic search mechanisms that are commonly used.

Symmetric Semantic Search: Symmetric semantic search involves queries and entries in the corpus that are similar in length and content. For example, a user may search for a question such as "How to learn Python online?" and seek to find a matching entry such as "How to learn Python on the web?". In this case, the query and corpus entries can be potentially flipped, making the search more efficient.

Asymmetric Semantic Search: In contrast, asymmetric semantic search typically involves a short query, such as a question or keywords, and the search is aimed at finding a longer paragraph or article that answers the query. For instance, a user may search for "What is Python?" and expect to find a paragraph that defines and explains the programming language. In this case, flipping the query and entries in the corpus is usually not meaningful.

Understanding the distinction between symmetric and asymmetric semantic search is crucial for designing an effective search system that delivers accurate and relevant results. By leveraging appropriate techniques, such as knowledge graphs, the semantic search can enable users to efficiently and effectively retrieve the information they need, regardless of whether the search is symmetric or asymmetric.

2.3 Knowledge Graph, Vector Database, and Vector Search

Knowledge Graphs [27] represent interconnected concepts and entities that are semantically related. On the other hand, Vector Databases [24] contain high-dimensional vectors that represent entities and concepts. Vector Search [21] is used to retrieve the most similar vector in the database to a given query vector. We will explore the fundamental concepts behind Knowledge Graphs, Vector Databases, and Vector Search in this section subsequently.

Knowledge Graph One of the most useful forms of knowledge representation is a knowledge graph. A Knowledge Graph is a graph-based knowledge representation that captures and organizes information in a structured format. It consists of nodes and edges, where nodes represent entities such as people, places, things, and concepts, and edges represent relationships between them. The relationships are typically labeled with semantic types, such as "is a," "part of," or "related to."

Knowledge Graphs enable the integration and linking of data from various sources and domains, which helps in knowledge discovery, question-answering, and decision-making. They are commonly used in applications such as search engines, recommendation systems, and intelligent assistants. For instance, the knowledge graph representation of the context of DevOps life cycle could be as follows. The nodes could represent various software tools and practices involved in the DevOps process, such as continuous integration (CI), continuous delivery (CD), and infrastructure as code (IaC). Edges could represent the relationships between these tools and practices, such as the fact that CI is a prerequisite for CD, or that IaC can be used in conjunction with containerization.

For example, the node for the CI tool Jenkins could be linked to the node for the version control system Git via an edge labeled "uses" indicating that Jenkins uses Git for source code management. Similarly, the Kubernetes container orchestration platform node could be linked to the Ansible configuration management tool node via an edge labeled "integrated with," indicating that Ansible can be used to manage the configuration of Kubernetes clusters.

Vector Database and Vector Search A Vector Database is a collection of high-dimensional vectors that represent various entities and concepts. Each vector corresponds to a feature or attribute of the entity and is represented by a numeric value. Vector Databases enable efficient retrieval and comparison of similar entities based on their features or attributes. They are commonly used in applications such as image and audio search, recommendation systems, and anomaly detection. Vector databases are powerful tools because they offer both traditional database functionality and the ability to search and compare vectors in an index.

Vector Search is the process of finding the most similar vector in a database to a given query vector. This is typically done by calculating the cosine similarity between the query vector and the vectors in the database. The cosine similarity measures the angle between the two vectors and is a widely used metric for vector similarity. With vector search, users can search for objects without needing to know their specific keywords or metadata classifications. Additionally, vector search can provide a wider range of results by returning matches that are similar or nearby, potentially revealing hidden items that might have been missed otherwise.

3 The CoDescribe Framework

In this section, we present our proposed framework for analyzing source codes and developing a question-answering system that can effectively respond to queries from development teams. This allows users to interact with the system and submit queries related to code explanations, code documentation generation, and error discovery. Overall, this framework represents a significant advancement in the field of software development, providing developers with a powerful new tool to effectively address various code-related queries that developers may encounter. The framework, as shown in Fig. 1, comprises three essential components: the *Embedding Handler*, the *QA System*, and a *Chat Interface* that facilitate communication with the developers.

In this section, we will provide an in-depth explanation of each component, highlighting their functionalities and features that make them integral parts of the framework.

3.1 Fine-Tuning Code Summarization Model

To develop a model for the code summarization process, it is essential to fine-tune the language model using a dataset of code descriptions. The CodeSearchNet dataset is a suitable choice for this purpose. The language model will undergo fine-tuning on this dataset to achieve optimal performance. We specifically choose CodeT5 [26] and LLaMA [25] models for this specific task. This fine-tuning process plays a significant role in achieving high-quality and accurate code explanations from the given source codes. Code summarization is very crucial to achieve our goal of creating

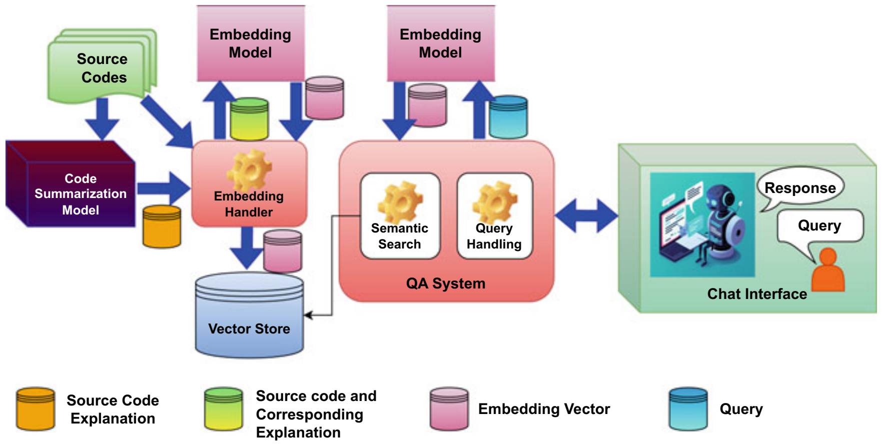


Fig. 1 The overall architecture of CoDescribe

a highly robust and accurate question-answering system that caters specifically to the needs of developers. The CodeT5 Model is a transformer-based language model specifically designed for code-related tasks. It is pre-trained on a massive amount of code and text data and can effectively understand and generate accurate code snippets, making it a popular choice for developers worldwide. On the other hand, the LLaMA model is an auto-regressive transformer-based language model that has been designed explicitly for generative tasks, such as text generation. It is an open-source model that has been trained on diverse data, which has led to its exceptional performance benchmarks across various Natural Language Processing (NLP) tasks, including Natural Language Understanding (NLU). Therefore, we have selected LLaMA as our preferred choice for the fine-tuning process. The CodeSearchNet dataset comprises a vast collection of code snippets and their corresponding explanations, making it an ideal training dataset to generate code explanations. Through the fine-tuning process, we can improve the CodeT5 model's ability to comprehend the source code and generate text in the form of explanations of the source codes.

Specification of Fine-tuning Process At this point, it is important to elaborate on the specifications of the fine-tuning process of the CodeT5 model on CodeSearchNet dataset.

- *Dataset*: CodeSearchNet [9] is a large-scale dataset that has been specifically designed for the task of code summarization, retrieval, and translation. It comprises more than 2 million code snippets written in six popular programming languages, namely, Java, Python, Ruby, Go, JavaScript, and PHP. The dataset is comprehensive and covers a wide range of programming tasks. Additionally, it includes several code-related features, such as function definitions, class definitions, and comments. Each code snippet in the dataset is associated with a set of natural language docstrings that describe the task the code snippet intends to accomplish. The docstrings used in the dataset are gathered from various programming forums and websites, which ensures that they represent the natural language used by developers when searching for code snippets. The dataset
- *Fine-tuning LLaMA and CodeT5 Model*: firstly, both the models are initialized with pre-trained weights on a large corpus of text. We fine-tune the model on the CodeSearchNet dataset by optimizing the model's parameters to minimize a loss function that measures the difference between the predicted output and the ground truth output. The optimization is done through backpropagation and gradient descent algorithms, where the gradients of the loss function with respect to the model's parameters are computed and used to update the model's weights. The hyperparameters of the models, such as the learning rate, batch size, and number of training epochs, are tuned to achieve optimal performance on the validation set. We provide summaries of hyperparameters that are used to train both the CodeT5 and LLaMA models in Table 2. Finally, the fine-tuned models are evaluated on the test set using smoothed BLEU-4 [20] score. Table 1 shows the performance of our code summarization models (marked with *) and compare with existing approaches.

Table 1 Smoothed BLEU-4 scores on the code summarization task

Method	Ruby	JavaScript	Go	Python	Java	PHP
CodeBERT	12.16	14.9	18.07	19.06	17.65	25.16
PLBART	14.11	15.56	18.91	19.3	18.45	23.58
RoBERTa	11.17	11.9	17.72	18.14	16.47	24.02
Fine-tuned CodeT5*	15.73	17.45	20.72	22.6	23.02	27.41
Fine-tuned LLaMA*	16.45	23.64	20.46	27.49	28.47	27.86

Table 2 Hyperparameter settings for training for CodeT5 and LLaMA models

Model	Learning rate	Train batch size	Evaluation batch size	Maximum sequence length	Epoch	Weight decay
CodeT5	3e-5	32	32	512	4	0.02
LLaMA	9e-5	64	64	512	5	0.02

3.2 Embedding Handler

The ability to understand a piece of text largely depends on how effectively it has been represented and how accurately the context has been captured through the embedding process. This highlights the importance of a robust and precise embedding approach for achieving optimal results in natural language processing tasks. The *Embedding Handler* module plays a crucial role in our proposed framework by utilizing embedding models to obtain embeddings for the source codes and their respective explanations. This module can handle different embedding models or embeddings providers like OpenAI, cohere, HuggingFace hub, TensorFlow hub, and many more. Another responsibility of this module is to efficiently store these embeddings, which can be achieved using various vector stores that also enable semantic search functionality. We summarize the input and output of this module as follows:

Input:

– Source code and their respective explanation.

Output:

– The module stores the embeddings in a Vector Database.

3.3 Chat Interface

Before discussing the module *QA System*, we must know about the inputs that are coming from the *Chat Interface*. This particular interface allows users to submit queries related to the source code they have previously supplied. Based on the nature of the query, the system generates a response that is then presented back to the user through the same interface. Hence, the *Chat Interface* serves as the gateway for user queries, while the *QA System* module functions as the intelligent component that generates the appropriate responses. We specifically mention the input and output of the module as follows:

Input:

- *External Input*: User query.
- *Internal Input*: Generated response from *QA System*.

Output:

- Carry forward the user query to *QA System*.
- Receive the generated response and send it back to the user.

3.4 QA System

The *QA System* is responsible for two major tasks to make the entire system functional.

Query Handling The *Query Handling* task obtains the embeddings for the query statement that is provided by the user. The embedding can be obtained using any language models capable of generating sentence embeddings. The popular sentence embedding models are T5 [22], Sentence BERT [23], models of GPT-3 [2] and GPT-3.5 series [19], GPT-J [3], GPT-NeoX [1], and many more. Some popular sentence embedding API providers are OpenAI,[1] Hugging Face Hub,[2] TensorFlow Hub, Cohere,[3] GooseAI[4], and many more. Inputs and outputs of the module are highlighted as follows:

[1] https://platform.openai.com/docs/api-reference.

[2] https://huggingface.co/docs/hub/index.

[3] https://docs.cohere.ai/docs.

[4] https://goose.ai/docs.

Input

- The query prompt given by the user.
- Embedding models or API.

Output

- Embedding of the query prompt.

Semantic Search The next most crucial task of the *QA System* is the semantic search. We imagine that our *QA System* needs to answer questions regarding the source code that we provide. To do that, we need a way to store and access that information when the *QA System* generates its response. In this context, the knowledge base comes into the picture. The knowledge base is a repository of information that can be semantically queried by our *QA System*. It is worth mentioning that, the *Embedding Handler* module has already stored the embedding vectors of the source codes and their corresponding explanations. The vector database creates indices for each document. This indices would allow us to quickly retrieve a list of relevant document based on a specific vector value or set of values that are similar to a given query vector. After creating our index, our *QA System* will be able to leverage to ground its answers in the relevant content to the user's prompts. The *QA System* internally uses the conversational model (GPT-3.5) and question-answering chain to generate appropriate response based on the search results. We summarize the inputs and output of the module as follows:

Inputs

- Embedding vector for the user query.
- Knowledge graph with stored vectors for codes and explanations.

Output

- Response phrase according to the search result from knowledge graph.

4 Evaluation of Experimental Setup

This section is dedicated to outlining the technological setup of the framework in great detail. We aim to give the internal workings of each module and explain why we chose specific technologies for each component. Our goal is to provide a comprehensive understanding of the technical details of the framework, enabling readers to gain a better understanding of its capabilities and how it works.

4.1 Prerequisites

The prerequisite for CoDescribe to be functional is the availability of the source codes and its corresponding explanations. In order to derive the code explanation, we fine-tune the LLaMA [25] model on CodeSearchNet [9] dataset. We specifically choose the LLaMA model as it is the generative language model and it sets a state-of-the-art performance for several works such as source common sense reasoning task, text summarization, text generation, code generation, and many more. LLaMA language model offers a distinct benefit in comparison to models such as GPT-3.5 (ChatGPT) or GPT-4 [17]. This advantage lies in its lightweight and open-source quantized version, which facilitates convenient fine-tuning using a standard GPU like Nvidia A100, and even without a GPU at all.

To explain further, once we derive the source code explanation, we utilize the GPT-3.5 [19] model to obtain embedding vectors for both the source code and its explanation. This specific model has been selected due to the ability of understanding subtle linguistic nuances, coherence, and context retention. The role of the *Indexer* module is to call the embedding model provider to generate embeddings for each document and then index those documents in Pinecone[5]. Whenever a vector is added to Pinecone, it is assigned an ID and indexed, which allows us to retrieve the vector quickly later on by querying the index. The index is essentially a data structure that organizes the vectors in a way that makes it efficient to search for them. Pinecone uses a variety of index types, such as Annoy, Hnsw, and Faiss, to support different use cases and trade-offs between retrieval speed and accuracy (Fig. 2).

4.2 Query Processing

When a user submits a query, it is first received by the *Query Builder* module. The *Query Builder* then generates a specific query based on the user's prompt and the conversation history. By doing this, downstream queries can take into account any relevant questions that the user has already asked. For instance, in the context of source code analysis, if the user asks "What is the purpose of *getData()* method?" and subsequently follows up with "How is this method called?", the *Query Builder* will understand the user's intent and generate a final inquiry such as "How is the method *getData()* invoked in the source code?". It is worth noting that, maintaining huge amount of conversation history is a difficult task and it slows down the query processing. To resolve this problem, conversation history is maintained by summarizing the conversation log over time and store it as conversation history. This approach ensures that the user's questions are interpreted in context and provides more accurate and relevant results. We also save the newly created query in our conversation history log.

[5] https://docs.pinecone.io/docs/overview.

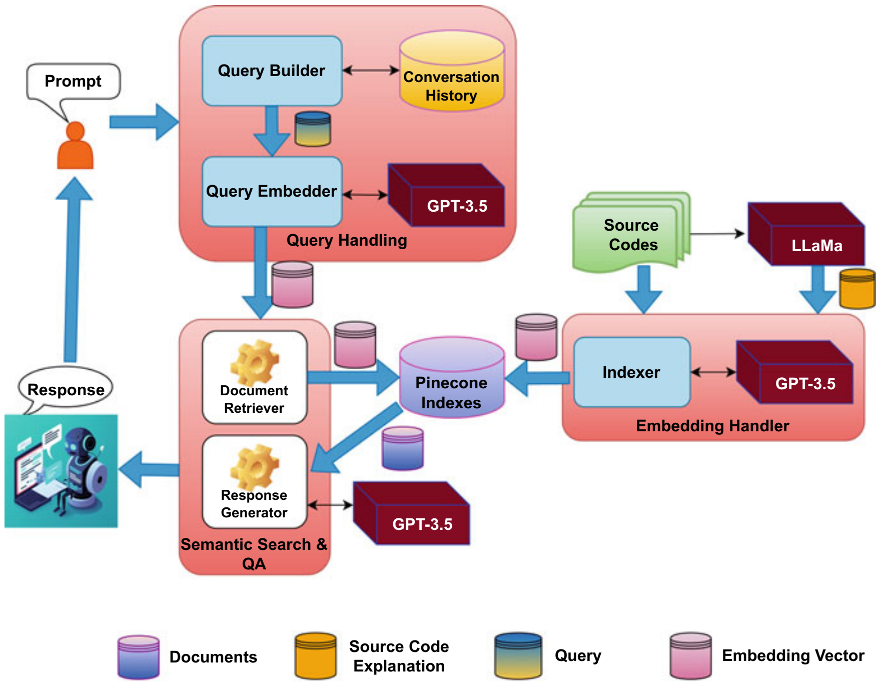


Fig. 2 Technology Stack of CoDescribe

After the *Query Builder* generates the query, the next step is to create an embedding vector for it. This allows the system to conduct a semantic search on the vector store. To generate the embedding vector, we utilize the OpenAI API[6] of the GPT-3.5 [19] model, which is designed to handle conversational text. It's important to note that our choice of embedding model is not restricted to a particular type of model. The framework is flexible enough to accommodate any sentence embedding models. However, since our system is designed for question-answering and conversational text, we have opted for a model that is best suited for this purpose.

4.3 Semantic Search

When a query is resolved and an embedding is generated, it will be used to query the Pinecone index which is populated by documents inserted by *Indexer* module. This will result in a number of potential hits, each with one or more corresponding documents from our source codes and their explanations. The *Document Retriever*

[6] https://platform.openai.com/docs/api-reference.

combines maximum marginal relevance mechanism with the semantic search mechanism to obtain most relevant documents from the Pinecone vector database.

Since these documents are most likely to be long, we create a summarizer chain to summarize long documents and produce a finalized summarized document that will be used to compose the final answer. The *Response Generator* utilizes the LangChain [18] library to create summarizer chain to summarize the documents retrieved from the Pinecone. The *Response Generator* also creates QA chain using LangChain and GPT-3.5 conversational model. By combining the summarizer and QA chains, the system is able to produce natural, human-like responses in a conversational question-answering format.

5 Analysis and Discussion

We perform a crowdsourced experiment with our framework on 6 use cases in this section. We take the help of 230 crowdsource agents in our evaluation of the framework's performance. It is worth mentioning that, we collect the feedback from the web application interface where user can provide the degrees of satisfaction for each of the use cases.

5.1 Specification of Crowdworkers

We collected the feedback from 220 undergraduate students who were in the third year (fifth semester) of their B.Tech. degree program in Computer Science and Engineering.[7] All the students had enrolled in Software Engineering courses and had successfully completed at least two software projects during their academic journey. To ensure a comprehensive analysis, we have also gathered feedback from 10 experienced industrial experts with over 5 years of experience in the field of software development. Their perspectives provided valuable insights into the practical aspects of software development, complementing the academic knowledge of the undergraduate students.

5.2 Aspects of the Crowdsourcing Experiment

We look into the following three aspects in order to conduct a precise analysis of the applicability, effectiveness, and acceptance of our framework based on the observations from our experiment (Table 3).

[7] B.Tech. students were from the Indian Institute of Information Technology, Vadodara, India and University of Calcutta, Kolkata, India.

Table 3 Degree of satisfaction for different use cases (Evaluated by Experts)

Satisfaction level	Satisfaction percentage for code explainability	Satisfaction percentage on documentation generation	Satisfaction percentage on error checking and code optimization	Satisfaction percentage on general queries on codes
Very satisfied	80	60	40	67
Satisfied	20	30	30	22
Average	0	10	20	11
Not satisfied	0	0	10	0

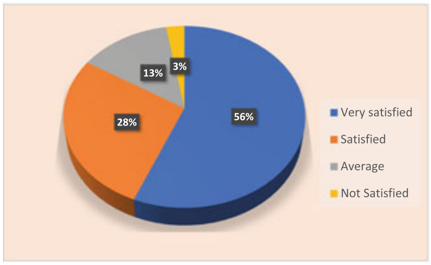
5.2.1 Applicability of the Framework to Software Engineers

We took feedback from 10 industrial experts about the applicability of the framework in terms of correctness and completeness of the generated outcomes. Assessing the applicability of the framework can be realized by analyzing the feedback of the industrial experts. After a thorough and extensive analysis on different use cases, they come to the following conclusion:

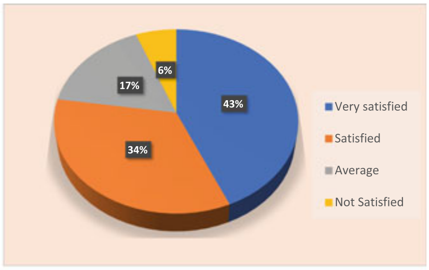
(i) *Evaluation on code explainability*: All 10 industrial experts agree on the generation of high-quality code explanation by the framework.
(ii) *Evaluation on documentation Generation*: 90% of the industrial experts are satisfied with the documentation generation from the different types of source codes.
(iii) *Evaluation on error correction and code optimization*: In this case, we find that 70% of the experts are satisfied with the error checking and code optimization task while 20% experts find that error correction and code optimization by the framework is in average state. On the same task 10% experts are not satisfied with the quality.
(iv) *Evaluation on General Query:* In this use case, 90% of the experts are satisfied, while 10% of them found it is in average state.

5.2.2 Benefits of Our Approach

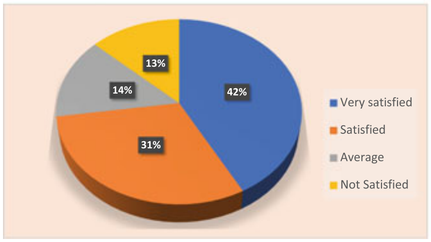
Based on the feedback from all the crowdworkers, our framework has demonstrated significant benefits for software engineers. As shown in Fig. 3, the generation of high-quality code explanations and documentation from different types of source codes has been unanimously praised by all the crowdworkers. This will be particularly useful for software engineers who need to understand complex codebases or work with unfamiliar programming languages. The framework's ability to correct errors and optimize code has also received positive feedback, indicating its potential to improve code quality and efficiency. However, our analysis reveals that 13% of crowdworkers are dissatisfied with this particular feature, which is a higher percentage compared to



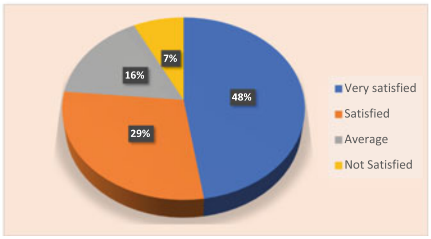
(a) Satisfaction on code explainability



(b) Satisfaction on documentation generation



(c) Satisfaction on error checking and code optimization



(d) Satisfaction on general queries on codes

Fig. 3 Degree of satisfaction for different use cases (Evaluated by Crowdworkers)

negative feedback received for other features. As a result, we have initiated further investigations into this feature to enhance it even further. Additionally, our framework introduces a chat interface that provides software engineers with a quick and easy way to get answers to their questions. This feature has received high satisfaction ratings from a majority of the crowdworkers, indicating its usefulness in increasing productivity and reducing development time. Notably, one of the key highlights of our tool, which enables reasoning across multiple source code files, underwent rigorous testing with the active participation of 161 out of 230 crowdworkers. The substantial positive feedback received combined with this substantial number of participants involved in this process underscores the success of this essential feature and positions our tool far ahead of existing solutions in the same domain. Overall, our approach has demonstrated significant benefits for software engineers in terms of improving code quality, increasing productivity, and facilitating collaboration between team members.

5.2.3 Feasibility in Terms of Time and Space

Our framework has demonstrated feasibility in terms of time and space due to its automated features that require minimal human intervention, saving significant amounts of time and effort. The tool support for the framework is hosted on a server with a single GPU and 16GB of RAM. It takes on an average 8 s to generate each response which is a decent response time with respect to such advanced NLP system. In terms of space, our framework requires minimal storage space (5GB) and can be easily integrated into existing software development workflows. Overall, our framework has demonstrated feasibility in terms of time and space, which is essential for its practical adoption in real-world software engineering projects. Despite of having these advantages, it is important to highlight two major concerns regarding this framework:

(i) *Time and effort required for index creation*: When dealing with large amounts of source code or complex projects, creating embedding vectors and organizing them in a vector store can be a time-consuming and resource-intensive task.
(ii) *Challenges of managing chat history*: Keeping track of lengthy conversations in real-time chat can be a challenging task, especially when dealing with a large volume of messages. To avoid this problem we manage the long conversation by summarizing the entire chat log, however this can also lead to a loss of context.

6 Threats to Validity

In this section, we list the limitations and other threats to the validity of our framework.

- *Generalization*: The main threat to validity is the ability of the framework to generalize to different programming languages and contexts, as this relies on the available elements that fit the technology stack. The models and techniques used in the framework may not be applicable to all programming languages, and may not be able to handle all types of queries or source code contexts.
- *Human feedback bias*: The use of human feedback to evaluate the applicability of the framework may introduce bias in the evaluation data, which could affect measuring the actual capability and applicability of the framework.
- *Evaluation metrics:* The evaluation metrics used to assess the performance of the framework may not accurately reflect its effectiveness or usefulness in real-world scenarios. For example, metrics, such as BLEU [20] or ROUGE [14], are designed for natural language summarization tasks, and may not be the most appropriate for evaluating the quality of code summaries or explanations.

7 Related Work

Software documentation and writing summaries are crucial for code comprehension and maintenance, but can be a tedious task for programmers. Automated code summarization can eliminate this burden and improve the quality and consistency of documentation. Prior researches [8, 13, 15] have explored various approaches to automated code summarization. These solutions have shown promise in reducing the burden of manual code summarization for developers to some extent. LeClair et al. [13] propose an ensemble strategy for neural code summarization that combines different approaches to improve performance. Their proposed method shows a performance increase of up to 14.8%, but the evaluation is limited to the BLEU [20] metric, and no comparison with state-of-the-art approaches is made. The paper suggests exploring more advanced aggregation strategies for future work. Krasniqi and Cleland-Huang [12] developed a tool to identify 12 types of code refactorings mentioned in commit messages, but encountered challenges due to inconsistencies in developer descriptions. The authors recommend future research on commit message recommender systems and integrating developers' and automatically generated descriptions for more complete and accurate descriptions. In another research [11], authors introduce CuBERT, a code-understanding BERT model pre-trained on a large corpus of Python files from GitHub. The authors create a benchmark of six program-understanding tasks to evaluate the performance of CuBERT and other models, and show that CuBERT outperforms other models, including state-of-the-art models. However, there are limitations, including the focus on only Python code, limited benchmark tasks, lack of interpretability analysis, and lack of detailed error analysis. Another paper [6] evaluates the effectiveness of local explanation methods on source code-based defect prediction models. It applies several popular methods and introduces automatic metrics to evaluate their effectiveness, but its limitations include the limited evaluation scope and not considering other local explanation methods or the impact of different programming languages and types of defects. Jain et al. [10] proposed an approach called ContraCode, a self-supervised representation learning algorithm that captures program semantics and is robust against adversarial attacks. The paper used an automated source-to-source compiler as a form of data augmentation to scalably generate variants for the proposed contrastive pre-training task. One limitation of the approach is that it relies on the availability of a source-to-source compiler, which may not be feasible in all scenarios. Nassif et al. [16] introduce a tool called Casdoc, a new format to improve the authoring and presentation of code examples. Casdoc embeds unobtrusive explanations into the code, links annotations to precise elements, and provides a need-oriented hierarchy of annotations. The authors produced 105 Java code examples and developed a prototype implementation for Java code examples that processes them and generates annotations containing API reference documentation. OpenAI has recently released a series of powerful language models, including the GPT-3.5-turbo, also known as ChatGPT, which is an advanced instruction-following model capable of performing a wide range of natural language processing tasks, such as code summarization, generation, and error checking in source codes. However, ChatGPT has some limitations when it comes to domain-specific tasks, such as the inability to handle multiple source code files and the problem of artificial hallucination.

In contrast, our proposed framework excels in handling multiple source code files and understanding their interconnectivity to provide accurate responses to user queries. Additionally, we have addressed the problem of artificial hallucination by forcing the model to rely on the given context to answer user queries. As a result,

Table 4 Comparison of existing code analysis techniques versus CoDescribe

Approach	Code summarization	Error finding	Code conversion	Documentation generation	Analysis of interconnected source codes
LeClair et al. [13]	✓	✗	✗	✗	✗
Krasniqi and Cleland-Huang [12]	✓	✓	✗	✗	✗
CuBERT [11]	✓	✓	✗	✗	✗
Gao et al. [6]	✓	✓	✗	✗	✗
Jain et al. [10]	✓	✗	✓	✗	✗
Nassif et al. [16]	✓	✗	✗	✓	✗
ChatGPT extension for IDE [19]	✓	✓	✓	✗	✗
CoDescribe	✓	✓	✓	✓	✓

if the reasoning required to answer a query is too complex, the framework will not provide an answer at all, reducing the chance of providing inaccurate responses.

In Table 4, we summarize the features of the above-mentioned state-of-the-art approaches and their limitations. The effectiveness of our approach can be appreciated by the comparison with them as depicted in the table.

8 Conclusion

The paper introduces a new framework that utilizes transformer-based models and knowledge graphs to perform question-answering-based tasks like code summarization and documentation generation. Furthermore, the framework answers complex user queries by reasoning, and supports error checking and code conversion to other languages. The contributions of this framework represent a significant advancement in the maintainability and explainability of software systems. We have performed an extensive crowdsource experiment to analyze the acceptability of the framework. We take the user feedback to access the satisfaction level of the overall experience. There are several potential directions for future work with respect to the proposed framework. One direction could be to investigate ways to improve the reasoning capabilities of the system by incorporating more external knowledge sources. In another direction, we aim to automate the process of augment codes to existing source code in order to implement additional requested features.

Acknowledgements Work partially supported by the EU-NGEU, iNEST-Interconnected NordEst Innovation Ecosystem funded by PNRR (Mission 4.2, Investment 49 1.5) Next Generation EU-Project ID: ECS 00000043, and by SPIN-2021 “Ressa-Rob” funded by Ca’ Foscari University.

References

1. Black S, Biderman S, Hallahan E, Anthony Q, Gao L, Golding L, He H, Leahy C, McDonell K, Phang J et al (2022) Gpt-neox-20b: an open-source autoregressive language model. arXiv preprint arXiv:2204.06745
2. Brown T, Mann B, Ryder N, Subbiah M, Kaplan JD, Dhariwal P, Neelakantan A, Shyam P, Sastry G, Askell A et al (2020) Language models are few-shot learners. Adv Neural Inf Process Syst 33:1877–1901
3. Chen M, Tworek J, Jun H, Yuan Q, Pinto HPDO, Kaplan J, Edwards H, Burda Y, Joseph N, Brockman G et al (2021) Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374
4. Cito J, Dillig I, Murali V, Chandra S (2022) Counterfactual explanations for models of code. In: Proceedings of the 44th international conference on software engineering: software engineering in practice, pp 125–134
5. Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D et al (2020) Codebert: a pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155
6. Gao Y, Zhu Y, Yu Q (2022) Evaluating the effectiveness of local explanation methods on source code-based defect prediction models. In: Proceedings of the 19th international conference on mining software repositories, pp 640–645
7. Guha R, McCool R, Miller E (2003) Semantic search. In: Proceedings of the 12th international conference on World Wide Web, pp 700–709
8. Haiduc S, Aponte J, Marcus A (2010) Supporting program comprehension with source code summarization. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering, vol 2, pp 223–226
9. Husain H, Wu HH, Gazit T, Allamanis M, Brockschmidt M (2019) Codesearchnet challenge: evaluating the state of semantic code search. arXiv preprint arXiv:1909.09436
10. Jain P, Jain A, Zhang T, Abbeel P, Gonzalez JE, Stoica I (2020) Contrastive code representation learning. arXiv preprint arXiv:2007.04973
11. Kanade A, Maniatis P, Balakrishnan G, Shi K (2020) Learning and evaluating contextual embedding of source code. In: International conference on machine learning. PMLR, pp 5110–5121
12. Krasniqi, R., Cleland-Huang, J (2020) Enhancing source code refactoring detection with explanations from commit messages. In: 2020 IEEE 27th international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 512–516
13. LeClair A, Bansal A, McMillan C (2021) Ensemble models for neural source code summarization of subroutines. In: 2021 IEEE international conference on software maintenance and evolution (ICSME. IEEE), pp 286–297
14. Lin CY (2004) Rouge: a package for automatic evaluation of summaries. In: Text summarization branches out, pp 74–81
15. McBurney PW, McMillan C (2015) Automatic source code summarization of context for java methods. IEEE Trans Softw Eng 42(2):103–119
16. Nassif M, Horlacher Z, Robillard MP (2022) Casdoc: unobtrusive explanations in code examples. In: Proceedings of the 30th IEEE/ACM international conference on program comprehension, pp 631–635
17. OpenAI (2023) Gpt-4 technical report. In: GPT-4 Technical Report

18. Ott S, Hebenstreit K, Liévin V, Hother CE, Moradi M, Mayrhauser M, Praas R, Winther O, Samwald M (2023) Thoughtsource: a central hub for large language model reasoning data. arXiv preprint arXiv:2301.11596
19. Ouyang L, Wu J, Jiang X, Almeida D, Wainwright C, Mishkin P, Zhang C, Agarwal S, Slama K, Ray A et al (2022) Training language models to follow instructions with human feedback. Adv Neural Inf Process Syst 35:27730–27744
20. Papineni K, Roukos S, Ward T, Zhu WJ (2002) Bleu: a method for automatic evaluation of machine translation. In: Proceedings of the 40th annual meeting of the association for computational linguistics, pp 311–318
21. Platzer C, Dustdar S (2005) A vector space search engine for web services. In: Third European conference on web services (ECOWS'05). IEEE, pp 9
22. Raffe C, Shazeer N, Roberts A, Lee K, Narang S, Matena M, Zhou Y, Li W, Liu PJ (2020) Exploring the limits of transfer learning with a unified text-to-text transformer. J Mach Learn Res 21(140):1–67. http://jmlr.org/papers/v21/20-074.html
23. Reimers N, Gurevych I (2019) Sentence-hert: sentence embeddings using siamese bert-networks. arXiv preprint arXiv:1908.10084
24. Stata R, Bharat K, Maghoul F (2000) The term vector database: fast access to indexing terms for web pages. Comput Netw 33(1–6):247–255
25. Touvron H, Lavril T, Izacard G, Martinet X, Lachaux MA, Lacroix T, Rozière B, Goyal N, Hambro E, Azhar F et al (2023) Llama: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971
26. Wang Y, Wang W, Joty S, Hoi SC (2021) Codet5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint arXiv:2109.00859
27. Wang Z, Zhang J, Feng J, Chen Z (2014) Knowledge graph embedding by translating on hyperplanes. In: Proceedings of the AAAI conference on artificial intelligence, vol 28 (2014)