Literature Review:

Fundamentally, summarization is the art of reducing a large amount of information into a concise and understandable format. The ability to condense lengthy texts into succinct but thorough summaries is essential for efficient communication and decision-making in a world where data is everywhere.

Natural Language Processing (NLP) algorithms are used in automated document summarization to evaluate and extract important information from texts. This mechanism looks for important words, phrases, or ideas based on criteria like importance and frequency. While this is important in document summarization, code base summary is a little different from text based ones.

Various Techniques applied:

1. DeepCom, NCS , and SIT are just a few of the many deep learning (DL)-based code summarization methods that have been proposed in recent years. They use potent generative models that were trained on a sizable corpus of code comments to convert short bursts of programming code into natural language summaries.
2. Information retrieval (IR)-based techniques
3. CodeT5 and CodeBERT work first to pre-train a large-scale language model (LLM) using tasks related to general language modeling, like unidirectional language modeling (ULM) and masked language modeling (MLM). Subsequently, the pre-trained models are refined for use in downstream tasks like code summarization, code search, and code clone detection.
4. ChatGPT is created by fine-tuning a GPT-3.5 series model via reinforcement learning from human feedback (RLHF). Leveraging three widely-used metrics (i.e., BLEU, METEOR, ROUGE-L) to measure the quality of the comments generated by ChatGPT compared to ground-truth comments. The experimental results (NCS, CodeBERT, and CodeT5) show that overall ChatGPT is inferior to the three SOTA code summarization models in terms of BLEU, METEOR, and ROUGE-L.

Various result:
https://paperswithcode.com/paper/codet5-identifier-aware-unified-pre-trained
This article shows the result codeT5 on various programming language and the current precision obtained by the model on them.

Table 6: Performance of ChatGPT and three SOTA models

Method	CSN-Python		
	BLEU	METEOR	ROUGE-L
NCS	15.8	10.6	31.3
CodeBERT	18.7	12.4	34.8
CodeT5	20.0	14.7	37.7
ChatGPT (one sentence)	10.28	14.40	20.81

Limitations of various models:

1.ChatGPT
Because code snippets are inherently random, ChatGPT produces different responses for the same code snippets when presented with different prompts. The possibility of misleading conclusions from random results is one risk to the validity of our study. Right now, ChatGPT is actively being developed. Our evaluation of ChatGPT is predicated on the GPT3.5 model.

2. DeepCom and other DL based models are code summarization techniques based on deep learning, and might run into issues typical to this field. The caliber and variety of the training dataset, possible sensitivity to particular programming languages or domains, and difficulties managing ambiguity in code could all have an impact on the system's performance.
3. Text summarization methods based on information retrieval (IR) have drawbacks because they depend on pre-established rules and patterns. Due to their tendency to give priority to surface-level features and predefined keywords, these approaches frequently fail to capture the semantic richness and context of text.

Current Approach required:
The two main drawbacks of the current code pre-training techniques are as follows. First, they frequently use models that are not ideal for generation and understanding tasks—encoder-only models like BERT or decoder-only models like GPT. For instance, when CodeBERT is used for the code summarization task, a second decoder is needed because the pre-training does not help this decoder. Second, most existing approaches treat source code as a sequence of tokens similar to natural language (NL), merely adopting the traditional NLP pre-training techniques. This misses a lot of the rich structural information found in programming languages (PL), which is essential to understanding the semantics of the code.

Since different models have different sets of frameworks under which it works, I suggest an ensemble model combining the base results of CodeT5 and CodeBERT to improve the performance on source code related data.

Comparison between CodeT5 and CodeBERT are given as:
1. **Model Architecture:**
CodeT5: CodeT5 is based on the T5 (Text-To-Text Transfer Transformer) architecture. T5 is a transformer-based model that treats all NLP tasks as a text-to-text problem. CodeT5 is fine-tuned specifically for code-related tasks, including code summarization and documentation generation.

CodeBERT: CodeBERT is based on the BERT (Bidirectional Encoder Representations from Transformers) architecture. BERT is designed for natural language understanding by pre-training on masked language modeling tasks. CodeBERT extends BERT for programming language-related tasks, including code summarization and code search.

2. **Training Strategy:**

CodeT5: CodeT5 follows the T5 approach, which involves pre-training on a large corpus of text data and then fine-tuning for specific downstream tasks, such as code summarization. The text-to-text framework allows CodeT5 to handle various NLP tasks in a unified manner.

CodeBERT: CodeBERT is pre-trained on a large-scale code corpus using masked language modeling, similar to the original BERT model. The pre-training process captures contextual information from code, enabling CodeBERT to understand and generate code-related content.

3. **Input Representation:**
CodeT5: CodeT5, following the T5 approach, treats both input and output as text sequences. It can take a code snippet as input and generate textual summaries or other text-based outputs.

CodeBERT: CodeBERT, being an extension of BERT, typically processes input code as sequences of tokens. It can capture bidirectional context within the code, making it effective for understanding the relationships between different parts of the code.

4. **Domain and Tasks:**
CodeT5: CodeT5 is designed to be versatile, handling various code-related natural language processing tasks, such as code summarization and code-to-code translation. Its text-to-text framework allows it to be applied to different tasks within the code domain.

CodeBERT: CodeBERT is specifically tailored for programming language-related tasks. It is often used for tasks like code summarization, code search, and understanding the semantics of code snippets.

5. **Fine-Tuning and Adaptability:**
CodeT5: The fine-tuning process for CodeT5 involves adapting the model to specific code-related tasks by providing task-specific labeled data. Its adaptability allows it to be customized for different code summarization scenarios.

CodeBERT: CodeBERT is fine-tuned for downstream tasks related to programming languages. The fine-tuning process involves training the model on task-specific datasets to adapt its knowledge to the nuances of code-related applications.

My perspective:

To use both of these good qualities, I suggest using experimented ensemble techniques due to the fact that they complement each other.

Based on the T5 architecture, CodeT5 is renowned for its adaptability to a variety of NLP tasks and is intended for use in text-to-text frameworks. Contrarily, CodeBERT uses BERT's bidirectional context to comprehend code semantics. A variety of representations that capture the combined benefits of bidirectional context understanding and the unified text-to-text

approach may be obtained by combining the two models. When it comes to managing various code patterns, CodeT5 and CodeBERT might be superior. Given its T5 architecture, CodeT5 might be effective at tasks that call for a variety of text-based representations, whereas CodeBERT might be better at capturing fine-grained code semantics due to its bidirectional understanding of code tokens. They may cover a wider range of code structures if combined.

Using an ensemble technique not only saves from noises and interventions specific to each code base but also captures the strong features from each model and use them to validate a better performance.

Implementation:

1.Create a code analysis module that uses language-specific parsers or abstract syntax trees (ASTs) to parse the input codebase. From the parsed code, locate and extract the names of the functions, methods, parameters, and code blocks. Accurately identify function structures by leveraging language-specific features.

2.To make sure the extracted functions are in a format that can be used as input into CodeT5 and CodeBERT, preprocess them. Tokenization, padding, and proper formatting of the code snippets may be required for this.

3. Load the tool with pre-trained CodeT5 and CodeBERT models. For a smooth integration, you can use the Transformers library from Hugging Face or other suitable libraries.

4. Create an ensemble strategy to integrate CodeT5 and CodeBERT outputs. Using a voting mechanism, averaging the logits or predictions, or even training a meta-model on top of the outputs of the individual models are some options.

5.To obtain individual summaries, run the preprocessed code snippet through CodeT5 and CodeBERT for each identified function. To create the final function summary, combine the summaries using the ensemble strategy. One way to do this could be to weight each model's contributions according to how well they've performed in the past.

6. Establish a system for evaluating quality so that the produced summaries can be evaluated. It is possible to measure the effectiveness of summarization using metrics like ROUGE scores or domain-specific evaluation criteria. This stage guarantees that the group method improves performance as a whole.

Ensuring that the tool comprehends the relationships between functions within a codebase is necessary to maintain coherence in function summaries. To create summaries that are meaningful and coherent, it is necessary to capture the context, dependencies, and interactions among functions which can be done using ASTs, and semantic understanding.

Limitations:

It is difficult for autonomous code summarization to deal with the variety of code structures and coding styles. If code summarization models are exposed to a variety of programming languages, coding paradigms, and individual coding styles, they may find it difficult to produce accurate and cohesive summaries. Creating accurate and contextually relevant summaries is

made more challenging by variations in variable naming, indentation, and commenting styles amongst codebases.

Possible solutions:
Creative solutions include adding a variety of code samples to training datasets that follow different coding conventions in order to address the problem of managing code variability and varying coding styles. Models can generalize more effectively when data augmentation techniques are used, such as introducing variations in coding styles during training. Moreover, adaptability to various styles is ensured by incorporating a pre-processing step that standardizes coding style elements prior to inserting code snippets into summarization models. Incorporating domain-specific knowledge and training models on cross-language datasets enhances the overall comprehension of varied coding practices, allowing autonomous code summarization tools to produce precise and cohesive summaries for a wide range of coding styles and languages.

References:

1. **https://arxiv.org/pdf/2308.14731.pdf**
2. **https://arxiv.org/pdf/2305.12865.pdf**
3. **https://blog.salesforceairesearch.com/codet5/**
4. **https://www.analyticsvidhya.com/blog/2023/07/exploring-gpt-2-and-xlnet-transformers/#:~:text=GPT%2D2%20for%20Text%20Summarization,-GPT%2D2%20is&text=It%20stands%20for%20(Generative%20Pretrained,relevant%20summaries%20of%20input%20texts.**
5. **https://paperswithcode.com/method/codebert#:~:text=Introduced%20by%20Feng%20et%20al,and%20natural%20language%20(NL).**