# Contextual Augmented Multi-Model Programming (CAMP): A Hybrid Local-Cloud Copilot Framework

Yuchen Wang, Shangxin Guo, and Chee Wei Tan,

arXiv:2410.15285v1 [cs.AI] 20 Oct 2024

*Abstract*—The advancements in cloud-based Large Languages Models (LLMs) have revolutionized AI-assisted programming. However, their integration into certain local development environments like ones within the Apple software ecosystem (e.g., iOS apps, macOS) remains challenging due to computational demands and sandboxed constraints. This paper presents CAMP, a multi-model AI-assisted programming framework that consists of a local model that employs Retrieval-Augmented Generation (RAG) to retrieve contextual information from the codebase to facilitate context-aware prompt construction thus optimizing the performance of the cloud model, empowering LLMs' capabilities in local Integrated Development Environments (IDEs). The methodology is actualized in *Copilot for Xcode*, an AI-assisted programming tool crafted for Xcode that employs the RAG module to addresses software constraints and enables diverse generative programming tasks, including automatic code completion, documentation, error detection, and intelligent user-agent interaction. The results from objective experiments on generated code quality and subjective experiments on user adoption collectively demonstrate the pilot success of the proposed system and mark its significant contributions to the realm of AI-assisted programming.

*Index Terms*—Article submission, IEEE, IEEEtran, journal, LaTeX, paper, template, typesetting.

## I. INTRODUCTION

THE field of natural language processing (NLP) has seen remarkable advancements through the use of large language models (LLMs). These models, capable of understanding and generating natural languages, have been fine-tuned to improve their performance using feedback mechanisms [1], [2]. The application of LLMs to AI-assisted programming has recently garnered significant attention [3], [4], as they offer the potential to embed advanced conversational agents into software development [5], [6]. This aligns with the visionary ideas presented in Edsger W. Dijkstra's seminal paper [7], illustrating the transformative potential of computers in facilitating a streamlined integration of code and human creativity.

The *MIT programmer's apprentice* was one of the earliest AI-assisted programming tools, aiming to simulate a knowledgeable junior programmer and utilizing NLP to understand programming patterns and interactions [8], [9]. This tool introduced revolutionary concepts such as code generation [10] and an early form of "prompt engineering" [11], driven by the recognition of computer programming as a systematic process of abstraction and simplification [7], [12].

AI-assisted programming improves software productivity by automating tasks, detecting errors, enhancing code quality, promoting usability, improving reliability, and accelerating the overall software development cycles [4]. Rather than replacing human programmers, these tools empower them to unleash
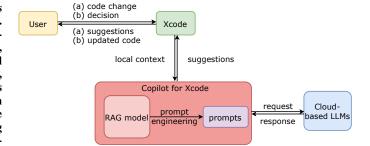


Fig. 1: An overview of *Copilot for Xcode*, the AI-assisted programming application that connects user requests (e.g., prompt tokens) with cloud-based LLMs leveraging context-based RAG.

their creative potential. By automating repetitive and mundane tasks, AI-assisted programming frees up valuable time and mental energy for human programmers to focus on innovative problem-solving and designing elegant solutions with the help of predictive analysis. Furthermore, by incorporating natural language processing capabilities (i.e., via prompt engineering), these tools enable human programmers to interact with software systems in a more intuitive and human-like manner, thus streamlining the software development process [13].

Cloud-based tools that leverage LLMs, such as Codeium [14], GitHub Copilot [15], OpenAI ChatGPT [16], and Amazon CodeWhisperer [17], enable users to access their cloud-based LLM services and online resources through dedicated application programming interfaces (APIs) in an on-demand manner. These tools can be incorporated into existing systems like a local integrated development environment (IDE) or implemented via a Software-as-a-Service (SaaS) web interface, acting as a virtual service entity to meet objectives and save costs for the human programmer [18]. The expanding reach and high demand usage of these LLM-based tools reflect the growing demand for advanced NLP capabilities in software development, resonating with Dijkstra's anticipation of a paradigm shift, where the major challenge lies not merely in the execution of programs but in the very process of their creation and maintenance [7], [13]. Dijkstra argued for the integration of documentation as an essential part of a program rather than just a guide for humans and envisioned the goal as treating documentation as an integral part of a program to address challenges related to malfunctioning, partial fault recovery, and modifying programs while in action [7].

The advent of Retrieval-Augmented Generation (RAG) has further revolutionized AI-assisted programming [19]. By com-

bining the strengths of pre-trained LLMs and information retrieval techniques, RAG models can retrieve relevant documents from a large corpus and use them to condition the generation process of the language model. This approach has shown great potential in enhancing programming tasks such as code completion, code generation, and error detection.

This paper presents a context-based RAG methodology that enhances LLM prompt construction with learned local contextual and content information. It also presents *Copilot for Xcode*, an AI-assisted programming tool that validates the proposed methodology in Xcode. *Copilot for Xcode* was open-sourced on December 7, 2022, one week after OpenAI launched ChatGPT on November 30, 2022. As shown in Figure 1, *Copilot for Xcode* seamlessly integrates cloud-based large language model services with local IDEs like Xcode. This integration benefits software developers in the Apple ecosystem by streamlining AI-assisted programming service delivery and enhancing the accessibility of a myriad of cloud-based LLM applications. *Copilot for Xcode* enables real-time prompt engineering and efficient interaction between the human programmer and the large language models, offering the potential to integrate serverless computing capabilities with natural language processing in the cloud.

The key contributions of this study include:

- We introduce CAMP, a novel multi-model approach utilizing context-based RAG to enhance AI-assisted programming with LLMs.
- We mathematically formulate the generalized problem of content retrieval and present algorithms to obtain the optimal retrieval model parameters.
- We present *Copilot for Xcode*, an actualization of CAMP on Xcode with innovative software solutions to overcome sandbox constraints in development environments like Xcode, which has been extensively utilized and freely available tool that integrates RAG and the OpenAI GPT series, providing benefits to thousands within the developer community.

## II. RELATED WORKS

### A. Retrieval Augmented Generation (RAG)

RAG is a recent development in the field of NLP that combines the strengths of pre-trained language models with information retrieval techniques. The RAG approach retrieves relevant documents from a large corpus and uses them to condition the generation process of the language model [19], [20], proposed initially as a method to leverage the vast amount of knowledge available in large text corpora to improve the performance in NLP tasks like question answering and fact verification. In the context of programming, RAG could potentially be used to improve the performance of code generation tasks by allowing the model to retrieve relevant code snippets from a large corpus of source code, which provides insights for our proposed work.

### B. Software Naturalness Hypothesis

The software naturalness hypothesis, articulated by [21], contends that programming languages should emulate the understanding and manipulation patterns inherent in natural language processing techniques applied to human languages. This hypothesis finds early substantiation in an n-gram model for code completion tasks, demonstrating the conception of software as possessing natural, repetitive, and predictable characteristics. The conceptualization of modeling codes through statistical language models with fine-tunable parameters inspires our proposed RAG module, detailed in Section IV-A, where we fine-tune hyperparameters, such as content search heuristics, to optimize the prompt engineering via maximum-likelihood estimation.

### C. Language Models for Big Code Analysis

LLMs have emerged as a promising approach to address challenges in computer programming, providing user-friendly means for constructing and modifying code with a natural language-like ease, thus vividly exemplifying the software naturalness hypothesis [21]. Since the introduction of the transformer architecture in 2017 [22], LLMs trained on large-scale datasets of programs have shown significant benefits in code-related tasks by effectively learning programming language patterns and structures, which are collectively part of Big Code analysis [23]. Recent LLMs such as T5 [24], BERT [25], GPT-4 [16] and Palm 2 [26] have demonstrated impressive capabilities in understanding and generating human-like text, opening up new possibilities for enhancing software engineers' development experiences.

### D. AI-assisted Programming

Dijkstra in [7] foresaw the growing complexity of high-level programming languages, anticipating the use of computer to assist human software engineers. His insight highlighted the importance of code translation, not just as a tool for human understanding, but as a crucial element in the code creation and modification process [7]. This foresight sets the stage for AI-assisted programming, which is the incorporation of machine learning techniques and tools into the software development process [27] to improve computer programming tasks. This concept shares similarities with pair programming [28], [29], whereby two human programmers collaborate to develop software by alternating between writing code (driver) and reviewing (observer) in a continuous switch. AI-assisted programming essentially replaces one of the two human programmers with an AI assistant, akin to the aforementioned *MIT programmer's apprentice* [8], [9]. The AI assistant automates tasks that can be broadly classified into two categories: generation and understanding. Generation tasks encompass activities such as code generation [30], [31], code completion [32], [33], code translation [34], [35], code refinement [36], and code summarization [37]. On the other hand, understanding tasks encompass activities like defect detection [38] and clone detection [39]. Improving the quality of large language models for these tasks focus on enhancing pre-training schemes [5], expanding training corpora [40], and employing improved evaluation metrics [6]. Many AI products have shown outstanding performances in coding assistance. AI-based predictive analysis [41] can anticipate
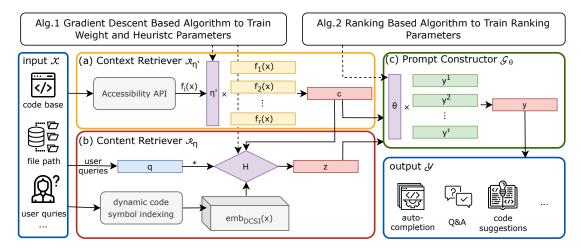
Fig. 2: Overview of the RAG module. (a) Context retriever $\mathcal{R}_{\eta'}$ that retrieves contextual information from the local development environment. (b) Content retriever $\mathcal{R}_\eta$ that searches for the most relevant information from local content. (c) Prompt constructor $\mathcal{G}_\theta$ that creates context-aware prompts.

potential issues in a software development life cycle, by proactively identifying and flagging critical incidents [42] before they manifest [43]. GitHub Copilot [15] and Codeium [14] are well utilized for code suggestions, ChatGPT [16] for code explanations, generation and natural language-based code modifications. However, few mature products have shown satisfactory performances to leverage LLMs in IDEs like Xcode, which makes our proposed work a necessity.

### E. Constraints of Local Integrated Development Environments (IDEs)

Local IDEs like Xcode [44] have been the cornerstone of software development for many years, providing a comprehensive set of tools for writing, debugging, and testing software. However, the integration of AI-assisted programming with LLMs remains challenging, due to overwhelmed computational demands [45], network latency and bandwidth limitations [46], and the isolated sandboxed environment [47]. These limitations represent significant hurdles to the effective application of LLMs in local IDEs, and addressing them is a key challenge for the field of AI-assisted programming.

These gaps in the related field mark the necessity of our proposed framework that integrates cloud-based LLMs into Xcode with context-driven RAG.

### III. PROBLEM FORMULATION

In this section, we mathematically formulate the language model that we require for AI-assisted programming, to explicitly define the problem metrics and show its feasibility.

Naturally, a "programming copilot" can be formulated as a language model with input (e.g. user commands, existing codes, past tokens, etc.) and output (e.g. automated completed content, suggested options, answers to user queries, etc.). If we zoom in on the context-based content generation that enhances AI-assisted programming with LLMs, the retrieval augmented generator we propose is formulated as a language model with input of "retrieved context" and output of "retrieved content".

We then construct the correlation between the context and content and solve for its solutions on the optimal parameters of the language model.

Let us start with the standard maximum entropy language model with the form [48]:

$$p(w|h) = \frac{\exp\left(a^T f(w,h)\right)}{\sum_{w'} \exp\left(a^T f(w',h)\right)}$$

where $w$ is the generated word given history $h$ and vector $a \in \mathbb{R}^d$ represents the model's parameters that are attached to the extracted features $f(w,h) \in \mathbb{R}^d$.

The model is easily extended to support separate "learning" of $w$ and $h$, with individual embeddings and a parameter matrix instead of the parameter vector

$$p(w|h) = \frac{\exp\left(\psi(w)^T A\phi(h)\right)}{\sum_{w'} \exp\left(\psi(w')^T A\phi(h)\right)} \tag{1}$$

where $\psi(\cdot) \in \mathbb{R}^{d_\psi}$ and $\phi(\cdot) \in \mathbb{R}^{d_\phi}$ are individual embeddings of the word and history and $A$ represents the extended parameter matrix.

A typical RAG model uses the input sequence to retrieve relevant content (also called "document") and then uses both the input and the document to generate the output sequence [19]. The retriever $p_\eta(z|x)$ computes the probability distribution of the top documents over the database, given input $x$; the generator $p_\theta(y_i|x, z, y_{1:i-1})$ then generates token $y_i$ based on the original input $x$ and the retrieved document $z$. The model is end-to-end formulated as

$$
\begin{aligned}
p_{\text{RAG}} &= \sum_{z \in \text{top-K}(p(\cdot|x))} p_\eta(z|x)p_\theta(y|x,z) \\
&= \sum_{z \in \text{top-K}(p(\cdot|x))} p_\eta(z|x)\prod_i^N p_\theta(y_i|x,z,y_{1:i-1}),
\end{aligned} \tag{2}
$$

and if each step of token generation can draw the distribution probability from all documents (which aligns with our case):

$$p_{\text{RAG}^t} = \prod_i^N \sum_{z \in \text{top-K}(p(\cdot|x))} p_\eta(z|x)p_\theta(y_i|x,z,y_{1:i-1}) \tag{3}$$

In our proposed context-based RAG module, the content retrial procedure is enhanced with context information in programming, such as file path and directory, current cursor position, and the user's point of view. While keeping the content representation $z$, we introduce an additional variable $c$ to represent the contextual information. The context variable $c$ then assists the context-based content retrieval

$$p_{\text{RAG}} = \sum_{z \in \text{top-K}(p(\cdot|x))} p_{\eta'}(c|x)p_{\eta}(z|x,c)p_{\theta}(y|x,c,z)$$

$$= \sum_{z \in \text{top-K}(p(\cdot|x))} p_{\eta'}(c|x)p_{\eta}(z|x,c) \prod_{i}^{N} p_{\theta}(y_i|x,z,y_{1:i-1}),$$

$$(4)$$

and similarly,

$$p_{\text{RAG}^{\text{t}}} = \prod_{i}^{N} \sum_{z \in \text{top-K}(p(\cdot|x))} p_{\eta'}(c|x)p_{\eta}(z|x,c)p_{\theta}(y_i|x,z,y_{1:i-1}).$$

$$(5)$$

From (4) and (5), the problem to solve can then be broken down to the modeling and optimization of individual sub-models $p_{\cdot}(\cdot|\cdot)$. Specifically, we need to handle the context retrieval model $p_{\eta'}(c|x)$, the content retrieval model $p_{\eta}(z|x,c)$, and the prompt generation model $p_{\theta}(y_i|x,z,y_{1:i-1})$. We dive into the details of the content retrieval model, as a typical example with relatively more complexity. We look back at (1) and cater it to our use case

$$p_{\eta}(z|x,c) = \frac{\exp\left(\psi(z)^T H \phi(x,c)\right)}{\sum_{z'} \exp\left(\psi(z')^T H \phi(x,c)\right)} \quad (6)$$

where $\phi(\cdot) \in \mathbb{R}^{d_\phi}$ extracts feature embeddings from both the original input and the context and $H \in \mathbb{R}^{d_\psi \times d_\phi}$ represents the heuristic matrix that determines the ranking of documents in the content search.

We finally show that this problem is feasible, has a global optimal solution, and can be solved in an iterative manner. We first claim the model defined by (6) to be in a continuous space, by taking the singular value decomposition of $H = U\Sigma V^T$. This indicates

$$p_{\eta}(z|x,c) = \frac{\exp\left(\psi(z)^T U\Sigma V^T \phi(x,c)\right)}{\sum_{z'} \exp\left(\psi(z')^T U\Sigma V^T \phi(x,c)\right)}$$

$$= \frac{\exp\left(\hat{\psi}(z)\Sigma\hat{\phi}(x,c)\right)}{\sum_{z'} \exp\left(\hat{\psi}(z')\Sigma\hat{\phi}(x,c)\right)} \quad (7)$$

where both $\hat{\psi}(z)$ and $\hat{\phi}(x,c)$ are continuous embeddings.

This can be formulated as a convex optimization problem

$$\min_{H} -\mathcal{L}(\mathcal{X},\mathcal{Y},H) + \mathcal{R}(H) \quad (8)$$

where $\mathcal{L}(\mathcal{X},\mathcal{Y},H) = \frac{1}{N}\sum_{i=1}^{N} \log P(y_i|x_i,H)$ is the target function and $\mathcal{R}(H)$ is the regularization term (e.g. $\ell_1$ norm, $\ell_2$ norm, nuclear norm, etc.). This can be resolved by the proximal gradient descent algorithm given by [49].

Our proposed work aims to answer the following questions:

- **RQ 1:** How to optimize AI-assisted programming with context-based RAG?

- **RQ 2:** How to leverage LLMs in sandbox-constraint development environments like Xcode?

We refer to the software naturalness hypothesis to give the mathematical definition of the research problems: compute a function $\mathcal{F}$ over the local development environment $x$, where the proposed model $\mathcal{M}$, with fine-tunable parameters $\gamma$, provides prompts for LLMs to finally obtain real-time "suggestions" $s$ as

$$\mathcal{F}_{\mathcal{M}_\gamma}(s|x) : \mathcal{X} \to \mathcal{S}$$

where $\mathcal{X}$, represents the domain of the input information, including environment-related information (e.g. source code, current repository, and editor status) and user-related information (e.g. detected user actions or proactively reported user requests); $\mathcal{S}$ represents the domain of the output "suggestions" provided by $\mathcal{M}$. Here, "suggestions" generally refer to any content revealed to the user that serves as programming assistance, including auto-completed code, error warning messages, answers to explicit requests in the chat panel, and so on.

Further, the model parameters $\gamma$ should optimize the maximum likelihood function $\mathcal{L}$ which measures the quality of results generated by the model $\mathcal{M}$ on the dataset $D$ used for model development as follows.

$$\hat{\gamma}_{\text{MLE}} = \arg\max_{\gamma} L(\mathcal{M}_\gamma(D)).$$

## IV. METHODOLOGY

In the next Section IV-A, we present our solution to the problem with $\mathcal{M}$ consisting of three sub-models: a contextual retriever, a content retriever, and a prompt constructor, and $\gamma$ consisting of the parameters corresponding to the models: $\{\eta', \eta, \theta\}$.

### A. Context-Based RAG

As mathematically defined in Section III, our proposed RAG module consists of three major components: (I) a context retriever $\mathcal{R}_{\eta'}(c|x)$ that captures contextual information from the input of local development environment, (II) a content retriever $\mathcal{R}_{\eta}(z|x,c)$ that generates top relevant content given the current context and the original input, and (III) a prompt constructor $\mathcal{G}_{\theta}(y_i|x,c,z,y_{1:i-1})$ that creates prompts to assist LLMs from the retrieved information and user queries. The RAG module is expected to support local context-aware AI-assisted programming, especially in mainstream tasks like code auto-completion and question handling.

Figure 2 presents a detailed illustration of the system workflow. Given the local development environment at a certain timestamp $t$, the contextual information $c$ is first obtained. Here we generally refer to input like code base and user queries as part of the local environment. The context $c$ is then utilized for the retrieval of the top-ranked relevant content information $z$, such as code snippets, interface, file path with programming structures, etc. Both the context and content are finally utilized in prompt construction for LLMs requests. Note that the local development environment is dynamically changing as $t$ changes, so the workflow is synchronous with

TABLE I: Major Components of a Constructed Prompt. Components are ranked in decreasing priorities which are computed with Algorithm 2. Content in the parameter denotes the model that mainly contributes to the entry.

| Component | Information Source | Priority |
|---|---|---|
| **Context System Prompt** ($\mathcal{R}_{\eta'}(c|x)$) | extracted from the retrieved local context | High |
| **Retrieved Content** ($\mathcal{R}_\eta(z|x, c)$) | obtained through content search | High |
| **New Message** | current communication entry | High |
| **Message History** | records of past interactions | Medium |
| **System Prompt** | defined based on user settings | Low |

user actions and code base changes, serving on-demand functionalities.

In the following subsections, we will explain the three components sequentially in detail.

*1) Context Retriever:* The mission of an effective context retriever is to pick from the boundless sea of local development environment the most representative drops that largely reflect the big picture and maximize the insights brought to the next step. Many factors in the input environment might be considered, intuitively, including the user's point of view, the current file opened, highlighted code snippets, and so on, though we can not afford to cover all possible aspects without "over-sparsing" the feature vectors or causing computational burdens. Without loss of generality, we define $\tau_c$ to be the upper limit of the contextual entries to include. We then have

$$\begin{aligned}
\mathcal{R}_{\eta'}(x) &= \mathrm{agg}([\eta'_0 c_0, \eta'_1 c_1, \ldots, \eta'_{\tau_c} c_{\tau_c}]) \\
&= \mathrm{agg}([\eta'_0 f_0(x_0), \eta'_1 f_1(x_1), \ldots, \eta'_{\tau_c} f_{\tau_c}(x_{\tau_c})]) \quad (9) \\
&= \mathrm{agg}(\eta' \cdot f(x))
\end{aligned}$$

where we abuse the annotation $f_i(\cdot)$ to represent the detailed data processing for each contextual entry and $\mathrm{agg}$ to represent the aggregation method. We make $\Sigma \eta'_i = 1$ for normalization and assign $\eta'_i$ a larger value to increase the influence of the corresponding $c_i$. $\eta'_i = 0$ for null entries when the number of selected components is lower than the limit $\tau_c$.

We gain insights from a typical work that helps blind programmers to effectively understand the structure of the code and navigate through the directories [50]. The proposed tool *StructJumper* presents a "TreeView" to programmers that contains anchor points that are tested to be most effective in helping people comprehend the coding structure, including methods and control flow statements. Our work shares a similar requirement to provide the most compacted contextual information to an agent who does not have full access to the environment.

We eventually select "cursor position" and "absolute repository path", "cached build artifacts", and "index information" as our sources of contextual information. We fine-tune the weight parameters $\eta'$ within the algorithm presented in Section IV-B. Note that we can arrive at a fixed set of optimal $\eta'$ across time and data $(\mathcal{X}, \mathcal{Y})$, with the assumption that the relative importance of different factors in the local development environment is stable.

*2) Content Retriever:* The objective of the content retriever is to deliver highly relevant content $z$ that effectively enhances the prompt construction with local context-aware information. This is also the gist of RAG: providing "documents" for generators to turn generalists into specialists. The retrieved

contextual information $c$ from the previous section plays two roles in this step: to support code base embedding and content search.

In (6), we have mentioned the usage of embeddings in content retrieval, where embedding functions $\psi(\cdot)$ and $\phi(\cdot)$ project the original sequences to low-dimensional embedding space for subsequent computation.

One-hot embedding is among the most lightweight methods, suitable for local tools and portable software. More advanced methods include neural network based encoders such as BERT [25], which are more advantageous in capturing complex patterns but have higher computational demands. To strike a balance between modeling power and computational efficiency, we propose dynamic code symbol indexing $\mathrm{emb}_{\mathrm{DCSI}}$ and employ it as the embedding function. This indexing method enables precise source code analysis by capturing each coding token's symbol information, position, relationships with neighboring tokens, and dependencies in the programming graph. It also achieves dynamic updates with changes like code base edits, staying in sync with the local context. While enabling efficient content search and code base content comparison by fully exploiting the contextual information, DCSI remains fast. This yields a simplified model

$$p_\eta(z|x, c) = \frac{\exp\left(\mathrm{emb}_{\mathrm{DCSI}}(z)^T H \mathrm{emb}_{\mathrm{DCSI}}(x)\right)}{\sum_{z'} \exp\left(\mathrm{emb}_{\mathrm{DCSI}}(z')^T H \mathrm{emb}_{\mathrm{DCSI}}(x)\right)} \quad (10)$$

where the consistent embedding function makes the heuristic $H$ a square matrix.

In certain scenarios, $H$ can be hard-coded and taken out as a constant. For instance, when we utilize the "cosine similarity", the exponential factor in (10) becomes

$$\frac{\mathrm{emb}_{\mathrm{DCSI}}(z) \cdot \mathrm{emb}_{\mathrm{DCSI}}(x)}{\|\mathrm{emb}_{\mathrm{DCSI}}(z)\|\|\mathrm{emb}_{\mathrm{DCSI}}(x)\|}.$$

Another baseline heuristic is the "location heuristic" where we rank content entries based on their distance to the current viewer point in the code graph. The location-weighted heuristic performs well in small-sized code bases due to the observation that shorter dependency paths usually signify robust correlations within code graphs.

We present the gradient descent algorithm to obtain the optimal values of $H$ (in the general form) and other parameters in Section IV-B.

Given a determinant embedding function and heuristic matrix, the content retriever identifies

$$\mathcal{R}_\eta(x, c) = \underset{z \in \mathrm{emb}(x)}{\mathrm{argmax}}\, p(c|H, q*)$$

where $q$ represents user query, which is optional because it is only explicitly given in certain cases like Q&A.

Listing 1: Structure of a Sample Constructed Prompt

```
 1  {
 2    "stream": true,
 3    "messages": [
 4      {
 5        "content": "You are an AI programming
              assistant. Your reply should be
              concise, informative, and logical
              .",
 6        "role": "system",
 7      },
 8      {
 9        "content": [FILE_PATH],
10        "role": "user",
11      },
12      {
13        "content": [CURSOR_POSITION],
14        "role": "user",
15      },
16      ...
17      {
18        "content": [GREETING_MESSAGE],
19        "role": "assistant",
20      },
21    ],
22    ...
23    "model": "gpt-35-turbo",
24    "temperature": 0.3
25  }
```

*3) Prompt Constructor:* The final component of the RAG module is the prompt constructor $\mathcal{G}(y_i|x, c, z, y_{1:i-1})$, where the retrieved context, content, and the interaction history with the user are then integrated into the new prompt.

Typically, a prompt is a JSON data structure that contains components in "key-value" pairs. Except for some mandatory fields such as "model" and "temperature", multiple components are configurable. Specifically, we can create components by specifying their values and arranging them in a certain order that represents their priorities. For example, we can embed the retrieved context and content to some components, as illustrated by the following toy example listing 1.

Table I presents some of the main components included in the constructed prompt. Each component is assigned a priority value that is used for ranking. Due to the limited context window, in cases where the provided content exceeds a certain length, components of lower priority values are truncated first. Remarkably, the order of messages is found to influence the output of the LLMs. For instance, when messages containing contextual information are placed after the message history, the LLMs tend to disregard the retrieved information and generate responses primarily based on the message history.

The objective of the prompt constructor is therefore to determine the optimal combination and ranking of the components. Denote the $i$th prompt as $y_i$ and the $k$th configurable component as $y^k$. Without loss of generality, let $\tau_k$ represent the maximum number of configurable components. Each $y_i$ is thus an ordered array of $y^k$. We consequently have

$$\mathcal{G}_\theta(x, c, z, y_{1:i-1}) = y_i$$
$$= \text{order}([y^1, y^2, \ldots, y^{\tau_k}]) \quad (11)$$
$$= \begin{bmatrix} \theta_1 & \theta_2 & \ldots & \theta_k \end{bmatrix}^T \begin{bmatrix} y^1 & y^2 & \ldots & y^k \end{bmatrix}^T$$

where $\theta_k$ are standard unit vectors that mark the component that is located on the $k$th position of $y_i$. We seek the optimal $\theta$ with an iterative algorithm presented in Section IV-B.

---

**Algorithm 1** Train Weight and Heuristic Parameters of Retrievers

**Require:** $\Sigma_i \eta'_i = 1, \theta_i$ are standard basis vectors
**Ensure:** $\tau_H > 0, \tau_{\eta'} > 0, \alpha^n > 0, \beta^n > 0$
$\quad H^1 = H^0 \in \mathbb{R}^{d_{\text{emb}} \times d_{\text{emb}}}, \eta'^0 = \eta'^1 \in \mathbb{R}^{d_c}$
$\quad c_0 \in \mathbb{R}^{d_c}, z_0 \in \mathbb{R}^{d_z}$
$\quad t^0 \leftarrow t^1 \leftarrow 1, n \leftarrow 1$
$\quad$**while** not converged **do**
$\qquad \bar{H}^n \leftarrow H^n + \frac{t^{n-1}-1}{t^n}(H^n - H^{n-1})$
$\qquad G^n \leftarrow \bar{H}^n - \frac{1}{\tau_H}\nabla_{\bar{H}^n}(\mathcal{L}(\mathcal{X}, \mathcal{Y}, H))$
$\qquad [U\Sigma V] \leftarrow \text{SVD}(G^n)$
$\qquad \bar{H}^n \leftarrow U\mathcal{D}_{\tau_H}(\Sigma)V^T \qquad \triangleright D_\tau X = \max(X - \tau, 0)$
$\qquad H^{n+1} \leftarrow H^n + \alpha^n(\bar{H}^n - h^n)$
$\qquad c = \mathcal{R}^{-1}_{\eta \supset H^{n+1}}(z)$
$\qquad \bar{\eta}'^n \leftarrow \eta'^n + \frac{t^{n-1}-1}{t^n}(\eta'^n - \eta'^{n-1})$
$\qquad g^n \leftarrow \bar{\eta}'^n - \frac{1}{\tau_{\eta'}}\nabla_{\bar{\eta}'^n}(\mathcal{L}(\mathcal{X}, \mathcal{Y}, \eta'))$
$\qquad \eta'^{n+1} \leftarrow \eta'^n + \beta^n(\bar{\eta}'^n - \eta'^n)$
$\qquad t^{n+1} \leftarrow \frac{1+\sqrt{1+4(t^n)^2}}{2}$
$\qquad n \leftarrow n + 1$
$\quad$**end while**

---

**Theorem 1.** *Starting from any initial weight $(\eta')$ and heuristic $(H)$ parameters of the retrievers, the optimal values can be obtained iteratively by Algorithm 1.*

*B. Parameter Tuning*

At this point, we have all the pieces of the context-based RAG module, except for some parameters to be tuned and optimized. This section presents the algorithm for model parameter tuning, including: 1) a gradient descent-based algorithm for computing the weight parameter $\eta'$ and heuristic matrix $H$ and 2) a sorting algorithm for computing the ranking parameter $\theta$.

---

**Algorithm 2** Train Ranking Parameters of Prompt Constructor

$\quad$initialize directional graph $G$
$\quad$**for** all possible $(\theta_i, \theta_j)$ **do**
$\qquad$**if** $\mathcal{L} - \mathcal{L}_{i \leftrightarrow j} > \epsilon$ **then**
$\qquad\quad$store directional edge $i - j$ to $G$
$\qquad$**end if**
$\quad$**end for**
$\quad$run topological sort on $G$
$\quad$**return** $G$

---

**Theorem 2.** *The optimal ordering of the $k$ prompt components $y^{1:k}$ can be solved by Algorithm 2 in $\mathcal{O}(k^2)$.*

To solve the optimization problem presented by (8), we introduce Algorithm 1 to train the weight and heuristic parameters of the retrievers iteratively, by alternatively moving $H$ and $\eta'$ to the negative gradient direction in each iteration $n$, with step size $\alpha$ and $\beta$ correspondingly. This algorithm is inspired by the Accelerated Proximal Gradient Algorithm

presented by [49] which utilizes the closed-form solution for minimizing the quadratic approximation of $f(X) + \|X\|_*$.

Given trained context and content retrievers, we obtain the individual prompt components $y^{1:k}$ as deterministic outcomes from the retrievers. We present Algorithm 2 to solve for the optimal ranking parameter $\theta$ as mentioned by (11). The question essentially is to find an arrangement of k items that yields the best result, so the brute-force method is to traverse all possible arrangements and return the optimal one, leading to $\mathcal{O}(k!)$ in time complexity. We observe that in the majority of cases, the difference brought by rearranging the order of a subset of the k components is trivial, and it is in certain cases we observe significant improvements in the result by switching two components. In other words, we can model the k components as k nodes in a directional graph where an edge represents the topological relationship between a pair of neighboring nodes. In Algorithm 2, we first test out the topological relationship between all pairs by switching them and comparing the result, and then run topological sort to arrive at a reasonable order, which leads to $\mathcal{O}(k^2 + k + C) \to \mathcal{O}(k^2)$ in time complexity where $C$ is the number of edges and is considered a relatively small constant in our use cases.

## V. Implementation Details

With the proposed RAG module which contains the trained retrievers and constructor, we are then ready to implement it in IDEs. The methodology can be generalized and applied to a broad range of local environments, but we specifically target ones with strict restrictions like sandbox constraints where the retrieved contextual information and content are necessary "single source of truth" about the development environment and play critical roles in enhancing AI-assisted programming.

This section presents the implementation details of our methodology in Xcode where we bridge the RAG module to both the local code base and the user to achieve real-time sync. As mentioned in Section II-E, extra efforts are required for IDEs like Copilot with sandboxed environments. For less restricted IDEs, the implementation becomes trivial.

### A. The Copilot for Xcode Framework

The structural framework of the *Copilot for Xcode* is illustrated in the sequence diagram presented in Figure 3. Upon users updating the code, the underlying model is promptly notified to retrieve information about the local context and user queries. Subsequently, it constructs prompts enriched with this contextual information, facilitating AI-assisted programming with LLMs. The response obtained is processed by the model and converted into suggestions for subsequent interactions between the user and the system, with *Copilot for Xcode* acting as the intermediary between the user and Xcode for code updates based on user feedback.

Throughout this operational sequence, *Copilot for Xcode* fulfills the following key functions: 1) local context retrieval, 2) prompt construction, 3) bridging to IDEs, and 4) user interactions. The first two functionalities address **RQ 1**, while the latter two are pertinent to **RQ 2**. This framework dynamically establishes a connection between local IDEs and users, integrating them with cloud-based LLMs.

### B. Bridging to IDEs

Enabling LLM functionality within Xcode is mainly constrained by two major obstacles: the IDE's sandboxing mechanism that delays synchronous user interactions and the limited information revealed by Xcode to software developers that hinders thorough code understanding. We correspondingly propose two key techniques to overcome these challenges and extract essential contextual information that is highly relevant to LLMs' prompt construction.

*1) XPC Service Level Communication:* Firstly, the Xcode's sandboxing mechanism restricts plugin access to specific resources and prevents the launching of other programs. Take the real-time code suggestion feature for instance: to utilize language servers like Github Copilot requires an additional program provided by Github to be executed alongside the plugin, posing a necessity for our system to bypass the sandbox of Xcode. To achieve this, we propose to establish communication between the Xcode source editor extension and a non-sandboxed XPC Service, which acts as a cross-process call service that facilitates the communication between the extension and the language server. This further allows the presentation of code suggestions in the user interface (UI) that is not constrained by Xcode.

*2) Accessibility API:* The second challenge is the limited permission allowed by Xcode to access local information, modify the code base, and interact with users. To assemble a request for language servers, sufficient information must be gathered from the development environment, but Xcode by default only provides the source code and file types. To obtain additional contextual information without relying on Xcode, we leverage the `Accessibility API` from the software development kit. This API captures and exposes information such as all text, cursor position, current editing file location, project location, etc, along with changes in the code base, enabling accurate local context retrieval. Besides the code base, the API also captures information about Xcode UI and supports interactions with the UI elements like the menu bar items. This empowers the proposed system to determine the appropriate location for content display and execute in-place code editing. We will present more details about user interactions in Section V-C.

*3) Local Context Retrieval:* The above techniques allow us to proceed with the local context retrieval and eventually achieve context-aware prompt construction. We retrieve local context $c$ from the development environment, including current cursor location, file path, and code snippets that users are paying attention to, from Xcode's data directory, as presented by (9). The directory is created by the Xcode development environment to store intermediate build and index information. It is separate from the project directory and is primarily used to cache build artifacts, compiled code, and index information generated during the development process.
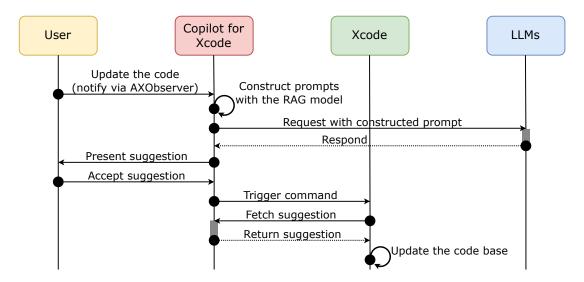
Fig. 3: Sequence Diagram of *Copilot for Xcode*. The sequence diagram illustrates the functionality of *Copilot for Xcode* which enables real-time suggestions through context-based RAG that leverages LLMs. *Copilot for Xcode* receives notifications upon code updates and subsequently constructs prompts with the proposed RAG module. Upon receiving the suggestions, the user has the option to adopt the recommendations and directly apply the changes to the code base.

## C. User Interactions

With the previous steps, we are able to construct context-aware and local content-sensitive prompts and leverage LLMs to obtain AI-assisted generated responses. The last piece of the puzzle thus lies in user interactions, where we detect real-time updates and deliver optimized content to users in the interface at proper timings.

We now introduce tasks that *Copilot for Xcode* completes. It facilitates seamless user interaction, offering high-quality code suggestions for selection, supporting real-time Q&A through an integrated chat panel, and enabling proactive updates to the local codebase based on user decisions. This comprehensive user interaction framework enhances the coding experience, ensuring developers can efficiently integrate suggestions and make informed decisions.

*1) Code Suggestion: Copilot for Xcode* offers dynamic and real-time code suggestions that adapt as users modify their codebase. Informed by the locally retrieved context, these suggestions are meticulously tailored to files currently open in the workspace, significantly improving accuracy. The system provides two presentation modes for displaying code suggestions. The "Nearby Text Cursor" mode, aligns suggestions suggestions with the current position of the text cursor, whereas the "Floating Widget" mode positions suggestions in proximity to the circular widget. This dual-mode presentation enhances the user experience by providing contextually relevant code suggestions based on the user's immediate coding environment.

Users' development experiences with code suggestions are further augmented by pre-defined commands. The `Get Suggestions` command retrieves customized suggestions based on the cursor position; the `Real-time Suggestions` automatically presents real-time suggestions in sync with the code repository; the `Prefetch Suggestions` command proactively fetches suggestions in the background for improved real-time responsiveness. Users can navigate through suggestions using `Next Suggestion` and `Previous Suggestion`, and making decisions with `Accept Suggestion` or `Reject Suggestion`. Crucially, users' decisions actively contribute to the assimilation of feedback, enhancing the customization of the service. The acceptance or rejection of code suggestions directly influences the locally retrieved context $c$, exerting potential influence on subsequent prompts $y$. This user-centric methodology ensures an adaptive and personalized code suggestion experience, aligning *Copilot for Xcode* with individual developer preferences and project-specific requirements.

*2) Chat:* To further enrich the intuitive coding experiences, *Copilot for Xcode* provides its chat and prompt-to-code features. These capabilities specifically focus on text-to-code generation and context-aware dialogues, customizing features to users' programming requirements. Users benefit from functionalities like extracting highlighted code from the active editor for reference, discussing specific code snippets, and effortlessly navigating through the codebase. Additionally, the feature expedites issue resolution by capturing errors and displaying warning labels in the active editor.

*3) Prompt-to-Code:* The prompt-to-code function extends a spectrum of capabilities for code creation and modification using natural languages. It caters to diverse use cases, including enhancing code readability, incorporating in-code documentation, and facilitating localizable string translation. Users can therefore seamlessly refactor existing code or compose new code by leveraging the expressive power of natural language.

These combined features empower users to engage in productive coding discussions and streamline their coding workflow, leveraging the prowess of RAG-enhanced AI-assisted programming.

## VI. EVALUATION

We evaluate the performance of *Copilot for Xcode* assisted programming with objective and subjective experiments, as a proof of concept of our proposed methodology. In the objective experiment, we run automatic code completion tests on a LeetCode database to measure the code quality; in the subjective experiment, we employ user studies to observe user adoption. Note that because of a lack of competitive tools on the same surface, our experiments mainly aim to ascertain the proposed tool's practical efficacy and gain insights into its applications in problem-solving and potential growth points for the future.

TABLE II: Statistics of the LeetCode Evaluation Database. The database collects algorithmic coding questions that are in Swift, each including a description of the functionality, a given function header, and an expected solution.

| Attribute | Value |
|---|---|
| Total # seed programs | 358 |
| Average # tokens per seed (rounded to unit) | 205 |
| Max # tokens among seeds (rounded to unit) | 722 |
| Min # tokens among seeds (rounded to unit) | 37 |

### A. Objective Evaluation

The objective evaluation assesses the performance of *Copilot for Xcode* through code completion tasks. Given a seed program, we divide it into two segments: the first part serves as a prompt, and the second part represents the "ground truth". During the evaluation, we conceal the second part and let the model complete the code based on the prompt. The generated code is then compared with the "ground truth" for performance analysis. As *Copilot for Xcode* represents the pioneering tool for Xcode, it lacks direct comparison models. The objective metrics employed in this study primarily serve as a Proof of Concept to glean valuable insights into the quality of the generated code.

*1) Experiment Setup:* In alignment with prevalent research practices in this domain [51] [52], we construct our evaluation dataset using a well-established repository of LeetCode solution programs in Swift [1]. LeetCode is a widely used online platform for honing coding skills, particularly for technical interviews. As reported by Table II, we took 358 algorithmic coding questions from the LeetCode repository as seed programs. Typically, each seed program includes a problem description and a main function that solves the problem. To form the prompt and "ground truth", we truncate the seed program at the first line following the problem description (excluding comments or blank lines). The model then generates code starting from the first token after this truncation point.

To achieve automated evaluation and mitigate potential biases introduced by manual testing, we create the simulation progress $P$ at the XPC service level to communicate with `CopilotForXcodeExtensionService`, a child process of *Copilot for Xcode* that manages code updates within Xcode.

[1]https://github.com/soapyigu/LeetCode-Swift

$P$ is linked to the LeetCode evaluation dataset, serving as a data provider for inputting test programs. This configuration enables the automated execution of code completion testing procedures.

TABLE III: Objective Evaluation Results on the Code Generation Task.

| Metrics | Results |
|---|---|
| Levenshtein Edit Similarity | 0.7418 |
| BLEU Score | 0.6849 |
| AST Normalized Similarity | 0.8796 |
| Semantic Similarity (RoBERTa) | 0.9914 |

*2) Results Analysis:* We utilize multiple metrics to evaluate *Copilot for Xcode*'s capabilities of code generation. The Levenshtein edit similarity [53] and BLEU score [54] are commonly adopted in related work for syntactic analysis of LLMs; the Abstract Syntax Tree (AST) Similarity quantifies the resemblance between ASTs of code snippets and is widely employed in evaluating code structural similarity. Additionally, we measure the similarity in contextual meaning with RoBERTa [55], by obtaining the semantic embeddings and computing the cosine similarity. This multifaceted approach aims to offer a comprehensive insight into *Copilot for Xcode* in generating code that aligns syntactically and structurally and semantically with the expected output.

Table III summarizes the objective results. Higher values in the metrics indicate greater content-wise similarity and structural resemblance between generated code and the "ground truth,". These results demonstrate the commendable performance of *Copilot for Xcode* in code completion tasks.

### B. Subjective Evaluation

We conducted user studies involving four participants to evaluate the practical usability of *Copilot for Xcode*, as inspired by the study that evaluates the usability of code generation tools at [56]. The participants were asked to solve different types of programming tasks with Xcode, both with and without the assistance of *Copilot for Xcode*. We closely observed their problem-solving processes and recorded key metrics, including time spent, bottleneck steps, and tool utilization.

*1) Experiment Setup:* The user study participants we found are university students and software engineers with at least one year of coding experience in Swift. Among the four participants (2 Female, 2 Male), 1 is an undergraduate, 1 is a master, and 2 are software engineers. For each participant, we organized an on-site user study session that lasted around one hour. During the session, participants were first briefed about how to use *Copilot for Xcode* and were given a short period ( 10 minutes) to freely explore the software. Then they were given two programming tasks to solve and asked a few questions afterward.

The two programming tasks we chose are one algorithmic and one UI-related problem:

- T1. Given an unsorted array, sort it with merge sort (without calling any APIs).
- T2. Create a "HomeView" and "DetailsView" with *SwiftUI* that navigates to each other.

To compare the impact of using *Copilot for Xcode* in solving these programming tasks while minimizing the learning effects, we randomly assign each participant to one of the four permutations of using vs not using *Copilot for Xcode* for the two tasks. The environment was set up in advance, with two laptops containing the folder of these 2 tasks in Xcode, one with *Copilot for Xcode* enabled and another not. For each task we assign a maximum 30 minutes and record "NULL" if participants failed to solve it within the time limit. To fully expose participants to the tool, we allow participants who are assigned to the control group (not using *Copilot for Xcode*) of a problem to still try the tool out after their performance was recorded.

After the tasks, they are asked the following open-ended questions:

- Q1. Do you feel more efficient in coding with *Copilot for Xcode*?
- Q2. How do you compare Xcode installed with *Copilot for Xcode* versus other AI portals like online search, or web-based GPT?

TABLE IV: Participants' Time Spent on Programming Tasks (In Minutes). The test group participants solve the questions with *Copilot for Xcode* and the control group not.

|  | Q1 Algorithm | | Q2 Swift UI | |
| --- | --- | --- | --- | --- |
| Group | Test | Control | Test | Control |
| Participant 1 | 15:21 | - | 8:06 | - |
| Participant 2 | 11:40 | - | - | 17:32 |
| Participant 3 | - | 20:48 | 12:20 | - |
| Participant 4 | - | 24:03 | 14:10* | NULL |
| **Average** | **14:00** | 22:25 | **10:13** | 17:32 |

*2) Results Anslysis:* Table. IV presents the time spent (in minutes) on the two programming tasks. A "NULL" value means that the participant failed to solve the problem within the given time limit. ∗ means that the participant is not assigned to that group but still tried it out after the assessment period. For both questions, the usage of *Copilot for Xcode* is shown to reduce the total completion time, spearing users from both coming up with the solution and manually typing out the code. Specifically, participant 4 originally failed to solve Q2 due to a lack of experience with *SwiftUI*, but was later able to build on the foundation provided by the tool and complete the task.

Regarding the open-ended questions, all four participants indeed feel more efficient with the assistance of *Copilot for Xcode* (and AI-assisted programming tools in general) while they are coding. For complex tasks, the tool usually provides a good framework that serves as a good starting point; for easier logic, the tool pushes them to move faster (e.g. by auto-generating content and implementations of popular algorithms). Compared to external AI-assisted programming methods, all four participants recognize the unique advantages of *Copilot for Xcode* which provides in-app support and user-centric interactions.

*3) Community Adoption:* To ascertain the practical quality and industrial values of *Copilot for Xcode*, we also publish its source code on GitHub [2]. This elevated the tool to a broader audience and encouraged collaboration and community involvement in its development. The GitHub repository received wide adoption, as indicated by the stars, forks, discussions, and contributions from the developer community, reflecting the tool's resonance within the developer community.

## VII. LIMITATIONS AND FUTURE DIRECTIONS

In this section, we critically analyze the limitations in our work and propose directions for future research works, in algorithm optimization, model refinement, and software features extensions.

*1) End to End Training:* The work we presented trains different components of the RAG module individually and sequentially, based on our assumptions of the data distributions (e.g. We potentially assume that the distribution $\eta'$ of importance of various context sources is independent of the rankings $\theta$ of different prompt components.). In future research, we propose to train all parameters end-to-end in one data pipeline, which might bring new insights to our understanding of the model parameters. However, this may also add to the computational burden and cause delays in user interactions.

*2) Trust AI:* The integration of data accessibility tools and generative AI services within the software introduces potential concerns and ethical considerations regarding data privacy.

To address this limitation and foster a user-centric approach, our next step involves the implementation of a robust user consent mechanism that empowers users with the ability to control the access of our tool to their development data and make informed decisions about the extent of information shared. We aim to closely align with emerging privacy regulations and frameworks such as the General Data Protection Regulation (GDPR) [57], to not only comply with legal requirements but also promote ethical data handling practices.

*3) Software Feature Extensions:* We also propose several extended features for *Copilot for Xcode* to prepare for its utilization in the production environment. As a software prototype, *Copilot for Xcode* overcomes considerable challenges during practical usage. For example, to bypass the sandboxing restrictions, it employs unconventional methods to retrieve local context information. As such, one future direction for us is to develop a portable functionality kit to maximize the tool's compatibility with future versions of Xcode. Secondly, as the current code suggestions are presented as C-style comments in comment mode, which can inadvertently disrupt a user's code if they are working on incompatible formats, we will work on extending to multiple programming languages for suggested code presentation. Furthermore, we will explore user coding preferences by exploiting user feedback during the interactions to improve customized services.

---

[2]https://github.com/intitni/CopilotForXcode

## VIII. Conclusion

This paper introduced a context-based RAG method that enhances AI-assisted programming with retrieved context-aware content from the development environment. It then proposed *Copilot for Xcode* that actualizes the methodology in Xcode, as a pilot success of AI-assisted programming in Apple's sandbox-constrained IDEs.

In this work, we explored the effectiveness of retrieval-augmented prompt engineering in AI-assisted programming and demonstrated its practical application in influencing code generation and guiding language models toward user-centric outcomes. We started with mathematically modeling the core problem as convex optimization problems and providing algorithms that arrive at globally optimal solutions with computational efficiency. We proceeded to implement the methodology in Xcode, with specific techniques to deal with the data constraint and bridge the tool with the local development environments. We then conduct preliminary objective measurements and subjective use studies as proof of concept which demonstrates the satisfactory performance of the proposed framework.

By combining the capabilities of context-based RAG and integrated tools for prompt engineering, *Copilot for Xcode* enhances and streamlines the software development process within Apple's Xcode. The integration of *Copilot for Xcode* with other cloud-based services like Xcode Cloud can also improve the overall productivity and efficiency in software development, which is especially important to continuous integration (CI) and continuous delivery (CD) in the software development pipeline. As AI-assisted programming tools like Copilot get incorporated into more IDEs, it brings us closer to the realization of Dijkstra's vision, fostering a symbiotic relationship between human programmers and AI-powered tools to achieve more efficient and reliable software development. The proposed methodology is also largely generalizable

## References

[1] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei, "Deep reinforcement learning from human preferences," *Advances in neural information processing systems*, 2017.

[2] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *Advances in Neural Information Processing Systems*, vol. 35, pp. 27 730–27 744, 2022.

[3] S. Rajamani, "AI assisted programming," in *15th Annual ACM India Compute Conference*, ser. COMPUTE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 5.

[4] M.-F. Wong, S. Guo, C.-N. Hang, S.-W. Ho, and C.-W. Tan, "Natural language generation and understanding of big code for AI-assisted programming: A review," *Entropy*, vol. 25, no. 6, p. 888, 2023.

[5] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu *et al.*, "Automating code review activities by large-scale pre-training," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022.

[6] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[7] E. W. Dijkstra, "A preliminary investigation into computer assisted programming," *E. W. Dijkstra Archive (EWD 237)*, (transcribed) 2007.

[8] R. C. Waters, "The programmer's apprentice: Knowledge based program editing," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 1, pp. 1–12, January 1982.

[9] C. Rich and R. C. Waters, "The programmer's apprentice: a research overview," *Computer*, vol. 21, no. 11, pp. 10–25, November 1988.

[10] R. E. Handsaker, "Code generation in the programmer's apprentice," MIT AI Lab, Working Paper 233, May 1982.

[11] C. Rich, H. E. Shrobe, R. C. Waters, G. J. Sussman, and C. E. Hewitt, "Programming viewed as an engineering activity," Massachusetts Institute of Technology, AI Memo 459, January 1978.

[12] C. Rich and R. C. Waters, "The disciplined use of simplifying assumptions," *ACM SIGSOFT Software Engineering Notes*, vol. 7, no. 5, pp. 150–154, December 1982.

[13] E. W. Dijkstra, "The humble programmer," *Communications of the ACM*, vol. 15, no. 10, pp. 859–866, 1972.

[14] E. Codeium, "Codeium - free AI code completion & chat," https://codeium.com/, 2023, accessed on June 1, 2023.

[15] N. Friedman, "Introducing github copilot: your AI pair programmer," 2021.

[16] OpenAI, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[17] C. Amazon, "AI code generator - amazon codewhisperer," https://aws.amazon.com/codewhisperer, 2022, accessed on June 1, 2023.

[18] L. Zheng, C. Joe-Wong, C. W. Tan, M. Chiang, and X. Wang, "How to bid the cloud," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2015, pp. 71–84.

[19] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[20] G. Izacard and E. Grave, "Leveraging passage retrieval with generative models for open domain question answering," *arXiv preprint arXiv:2007.01282*, 2020.

[21] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *2012 34th International Conference on Software Engineering*. IEEE, 2012, pp. 837–847.

[22] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in Neural Information Processing Systems*, 2017.

[23] M. Vechev, E. Yahav *et al.*, "Programming with "big code"," *Foundations and Trends® in Programming Languages*, vol. 3, no. 4, pp. 231–284, 2016.

[24] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *The Journal of Machine Learning Research*, 2020.

[25] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[26] R. Anil, A. M. Dai, O. Firat, M. Johnson, D. Lepikhin, A. Passos, S. Shakeri, E. Taropa, P. Bailey, Z. Chen *et al.*, "Palm 2 technical report," *arXiv preprint arXiv:2305.10403*, 2023.

[27] H. Mozannar, G. Bansal, A. Fourney, and E. Horvitz, "Reading between the lines: Modeling user behavior and costs in AI-assisted programming," *arXiv preprint arXiv:2210.14306*, 2022.

[28] C. Bird, D. Ford, T. Zimmermann, N. Forsgren, E. Kalliamvakou, T. Lowdermilk, and I. Gazit, "Taking flight with copilot: Early insights and opportunities of AI-powered pair-programming tools," *Queue*, vol. 20, no. 6, pp. 35–57, 2022.

[29] S. Imai, "Is github copilot a substitute for human pair-programming? an empirical study," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 319–321.

[30] R. J. Waldinger and R. C. Lee, "Prow: A step toward automatic program writing," in *1st International Joint Conference on Artificial Intelligence*, 1969, pp. 241–252.

[31] Z. Manna and R. J. Waldinger, "Toward automatic program synthesis," *Communications of the ACM*, vol. 14, no. 3, pp. 151–165, 1971.

[32] R. Robbes and M. Lanza, "How program history can improve code completion," in *23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 317–326.

[33] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *7th Joint Meeting of The European Software Engineering Conference and The ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2009, pp. 213–222.

[34] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API patterns as partial orders from source code: From usage scenarios to specifications," in *6th Joint Meeting of The European Software Engineering Conference*

*and The ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2007, pp. 25–34.

[35] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Association for Computing Machinery, 2014, p. 281–293.

[36] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object-oriented program repair," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2017, pp. 648–659.

[37] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Generating parameter comments and integrating with method summaries," in *2011 IEEE 19th International Conference on Program Comprehension*. IEEE, 2011, pp. 71–80.

[38] E. Charniak, *Statistical Language Learning*. MIT press, 1996.

[39] K. A. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein, "Pattern matching for clone and concept detection," *Automated Software Engineering*, vol. 3, no. 1-2, pp. 77–108, 1996.

[40] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[41] Y. Ji, A. Bosselut, T. Wolf, and A. Celikyilmaz, "The amazing world of neural language generation," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Tutorial Abstracts*, 2020, pp. 37–42.

[42] N. M. S. Surameery and M. Y. Shakor, "Use chatgpt to solve programming bugs," *International Journal of Information Technology & Computer Engineering (IJITC) ISSN: 2455-5290*, vol. 3, no. 01, pp. 17–22, 2023.

[43] K. Talamadupula, "Applied AI matters: Ai4code: Applying artificial intelligence to source code," *AI Matters*, vol. 7, no. 1, pp. 18–20, 2021.

[44] X. Apple, "Xcode 15 - apple developer," https://developer.apple.com/xcode/, 2003, accessed on June 1, 2023.

[45] V. J. Hellendoorn, P. T. Devanbu, and A. Bacchelli, "Will they like this? evaluating code contributions with language models," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 157–167.

[46] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[47] Apple Inc. (2021) App sandbox. Accessed: October 22, 2024. [Online]. Available: https://developer.apple.com/library/archive/documentation/Security/Conceptual/AppSandboxDesignGuide/AboutAppSandbox/AboutAppSandbox.html

[48] R. Rosenfeld *et al.*, "A maximum entropy approach to adaptive statistical language modelling," *Computer speech and language*, vol. 10, no. 3, p. 187, 1996.

[49] K.-C. Toh and S. Yun, "An accelerated proximal gradient algorithm for nuclear norm regularized linear least squares problems," *Pacific Journal of optimization*, vol. 6, no. 615-640, p. 15, 2010.

[50] C. M. Baker, L. R. Milne, and R. E. Ladner, "Structjumper: A tool to help blind programmers navigate and understand the structure of code," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, 2015, pp. 3043–3052.

[51] N. Nguyen and S. Nadi, "An empirical evaluation of github copilot's code suggestions," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 1–5.

[52] Z. Li, C. Wang, Z. Liu, H. Wang, D. Chen, S. Wang, and C. Gao, "Cctest: Testing and repairing code completion systems," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1238–1250.

[53] V. I. Levenshtein *et al.*, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8. Soviet Union, 1966, pp. 707–710.

[54] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[55] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[56] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Chi conference on human factors in computing systems extended abstracts*, 2022, pp. 1–7.

[57] European Parliament and of the Council. (2016) Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/ec (general data protection regulation). [Online]. Available: https://eur-lex.europa.eu/eli/reg/2016/679/oj