Automated Code Generation Using AST and RAG for Software Components of Microcontrollers

Sebastian Haug
Munich University of Applied Sciences,
Munich, Germany
shaug@hm.edu

Prof. Dr. Christoph Böhm
Munich University of Applied Sciences,
Munich, Germany
christoph.boehm@hm.edu

Daniel Mayer
Executive Director, AGSOTEC GmbH,
Munich, Germany
daniel.mayer@agsotec.com

***Abstract*—This paper proposes a method that can generate complete software components for embedded systems, fitting seamlessly into existing implementations without developer intervention. We demonstrate this approach by automatically generating hardware abstraction layer (HAL) code for GPIO operations on the STM32F407 microcontroller. The proposed method utilizes Abstract Syntax Trees (AST) to analyze existing code and Retrieval-Augmented Generation (RAG) to produce the missing components, effectively enabling code completion for embedded applications.**

***Index Terms*—Automated code generation, AST, RAG, Embedded systems**

I. Introduction

In recent years, the demand for embedded software has been on the rise due to advancements in industries like IoT and machine learning [1].

A. Problem

The progress of LLms has made Code-Generation with AI practicable. For now Large language models (LLMs) require software developers to provide prompts that are often contextually dependent. These models cannot autonomously generate working code without specific developer input regarding the missing program parts and their intended usage.

B. Problem-Solution-Fit

The goal of this paper is to explore how to identify interfaces that lead to missing code and the context in which the absent code is used. The idea is to automate this process to allow for generation of software components without extensive manual guidance. Specifically generating the hardware-specific code which connect the application-layer with the hardware-operations.

C. Motivation

Code generators have proven useful because they allow for rapid development of similar software components. However, these tools are often costly and complex to create, limiting their widespread adoption.

D. Goals of this paper

The goal of this paper is to implement a method or process to create simpler code generators.

E. State of the Art

- What Copilot can already do - What Auto Bug fixing Tools can already do

Conclusion "Competitor" Research: - What do current solutions Lack? - a systematic approach to generate misssing code-parts for microcontrollers.

Goal is it to utilize existing development methodologies and tools such as AST[1] and RAG[2] to facilitate the creation of missing program elements.

F. Outline

1. Background
2. Prototypical Implementation
3. Experimental Evaluation
4. Discussion/Interpretation
5. Threats to Validity
6. Related Work
7. Outlook

II. Background

A. Hardware Abstraction Layer (HAL)

The Hardware Abstraction Layer (HAL)[3] provides a standardized interface for interacting with hardware, abstracting away hardware-specific details and complexities [2]. This layer allows higher-level software to communicate seamlessly with hardware components without requiring intimate knowledge of the underlying hardware. The HAL thus promotes code portability and reuse across different hardware platforms by offering a uniform interface for common hardware operations.

In Figure 3, we provide a visualization of the simplified code structure generated by our approach. This visualization highlights the distinct layers in our implementation:

[1]AST: Abstract Syntax Tree
[2]RAG: Retrieval-Augmented Generation
[3]HAL: Hardware Abstraction Layer

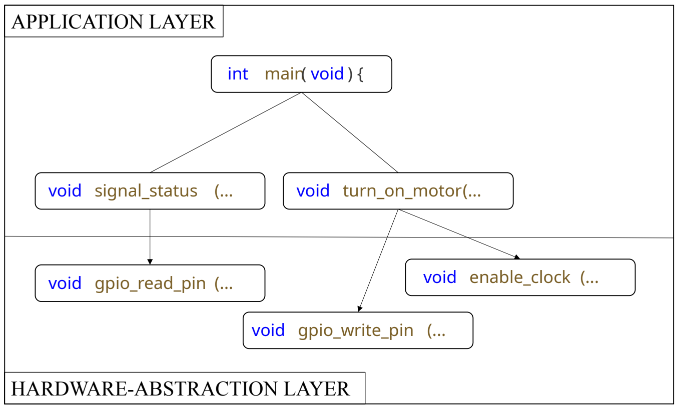


Fig. 1: **AST Implementation Process.** Visualization of the layers involved in code generation, showcasing the Application Layer and Hardware Abstraction Layer (HAL) interactions.

At the top of the hierarchy is the **Application Layer**, where the main entry point of the program is defined, e.g., `int main(void)`. In this layer, high-level functionality—such as controlling external devices, like turning on a car's rear light—is implemented.

Below the application layer is the **Hardware Abstraction Layer (HAL)**. Here, hardware-specific operations, such as enabling a clock or reading and writing to GPIO pins, are performed. The application layer interacts with the HAL by calling these hardware-specific functions. For example, to enable the clock for GPIO, read the pin state, or write to a GPIO pin, we generate the necessary HAL functions like `ENABLE_GPIOA_CLK()`, `HAL_gpio_read()`, and `HAL_gpio_write()`.

This clear separation of layers allows us to abstract away hardware details while maintaining efficient control over low-level operations.

The complexity of configuring GPIO ports and understanding the underlying hardware specifics highlights the challenges involved in embedded system engineering [3]. Such configurations require precise control and thorough knowledge of both the hardware and its interaction with software.

B. Code-Generation Techniques

In this paper, we use advanced code-generation techniques to enhance Large Language Models (LLMs) by integrating them with context-specific information from existing codebases and documentation. A significant challenge in generating hardware abstraction layer (HAL) code is not only creating syntactically correct code but also ensuring that the generated code fits seamlessly into the larger codebase where it will be applied. To solve this, we use a LLM with to generate code and vectorstores to retrieve relevant contextual usage examples, ensuring that the generated HAL code integrates accurately and efficiently with existing systems.

1) Embeddings and Vector Stores: Embeddings form the backbone of our RAG approach by representing both natural language and code as dense vectors in a high-dimensional space. Originally introduced by Mikolov et al. [4], embeddings capture semantic relationships between different pieces of data, mapping similar meanings to nearby points in the vector space. This capability allows the system to comprehend and retain contextual information from the codebase in a structured format.

By leveraging embeddings, we can represent code snippets and documentation as vectors that reflect their functional and semantic meaning. This enables the model to not only generate code but also understand how that code should be used within the specific context of a larger project. Storing these embeddings in vector stores provides an efficient way to retrieve related code segments and contextual data that guide the LLM during code generation.

The use of vector stores allows the system to perform fast and accurate lookups, retrieving relevant code snippets from large datasets based on their similarity in the embedding space. This process is crucial for our RAG framework, as it provides the LLM with examples and references that ensure the generated HAL code is both contextually appropriate and ready for integration into the codebase.

a) FAISS: Facebook AI Similarity Search: To handle large-scale embedding searches, we utilize FAISS (Facebook AI Similarity Search), introduced by Douze et al. in 2024 [5]. FAISS is a powerful tool for indexing and searching high-dimensional vectors, designed to handle the trade-off between search accuracy, speed, and memory efficiency. It is particularly well-suited for real-time code generation tasks that rely on fast retrieval of relevant data from extensive codebases.

FAISS enables the system to scale across hardware, from CPUs to GPUs, allowing it to process and search through trillions of vectors with high efficiency. This is critical for our RAG system, as it ensures that the generated HAL code is informed by how similar code is used elsewhere in the codebase, improving both accuracy and integration.

III. Prototypical Implementation

A. Microcontroller Selection

To validate the methods proposed in this paper, we primarily focus on the STM32F407 microcontroller, which is widely used in various embedded systems due to its robust features and broad support in the developer community [6].

The STM32F407[4] provides a balanced combination of performance, power consumption, and peripheral support, making it an ideal candidate for developing and testing HAL generation methods.

B. Integration of GPT-4o Mini

We integrate the GPT-4o Mini[5] into this study, because of the model's affordability ensures that resources are sufficient for the iterative testing of the researched techniques.

[4]STM32F407: High-performance Microcontroller
[5]GPT-4o Mini: Large Language Model from OpenAI

1) Prompt Engineering: Prompt Engineering is essential for leveraging Large Language Models (LLMs) to automate the generation of software components in microcontroller development. As described by Urban in his works on prompt engineering techniques [7, 8], improving the design of prompts enables developers to effectively utilize LLMs[6] to produce efficient, reliable, and maintainable code tailored to the specific requirements of embedded systems.

A structured prompt used in this paper includes several key components:

a) Cue: The cue defines the function or identity that the automated system or tool will assume while executing the prompt. It sets the expectation and scope of the tasks that the system will handle, which is crucial in effective prompt design [7].

```
You will be my Custom Hardware Abstraction Layer
    Generator.
```

b) Clear Instructions: Starting with clear instructions is vital, as it sets the foundation for the task at hand. Urban emphasizes that the sequence of information in the prompt significantly affects the model's response [8]. Clear instructions include both a clear goal and constraints.

```
Please generate a custom C function
    implementation for the function '{
    function_name}' with {length_parameters}
    parameters like: {sample_parameters}.
```

c) Constraints: constraints are important

```
Don't s:
- Don't reference new variables or functions that
     are not implemented.
- Don't reference stm32fxxx_hal.h functions.
...
```

Fig. 2: **Constraints in Prompt Engineering [8].**

d) Return Format: This specifies how the results of the prompt should be structured and presented, enhancing the usability and clarity of the generated content [9].

```
Return-Format:
...
- Be well-documented with comments explaining its
     purpose, parameters, and return value.
- Create your own custom HAL functions without
    referencing other functions.
...
```

[6]LLM: Large Language Model

e) Supporting Context: Context includes any background information, specific scenarios, or data that needs to be considered while executing the prompt. This helps tailor the response to fit the specific needs or circumstances of the project [7].

```
Create the {function_name} using the provided
    information about the existing code for an
    STM32F407 board: {context}
```

2) Detecting missing Code-Elements: One of the objectives is to enable a program to automatically build upon an existing codebase. Achieving this requires the program to have an understanding of the code structure, including the relationships between functions, variables, and dependencies. To facilitate this, we employ an abstract syntax tree representation of the program code to analyze and navigate the code effectively [10].

These method combined with LLMs and vector stores can automate the generation of the HAL. The process developed uses these techniques to extract and analyze functions and variables, identify any missing elements, and generate the necessary code to fill these pieces.

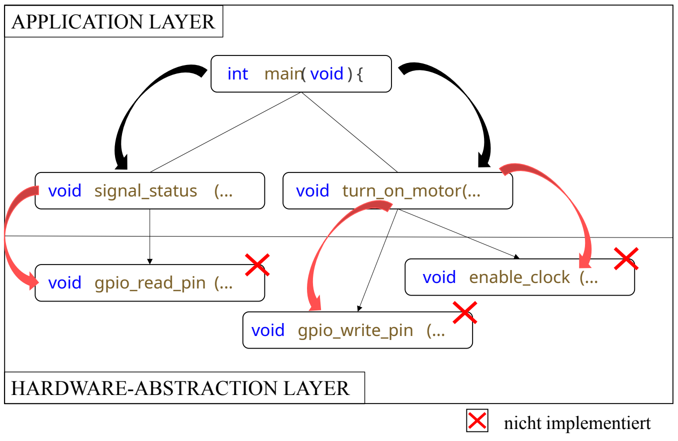


Fig. 3: **AST Implementation Process.** This figure illustrates the process of using an Abstract Syntax Tree (AST) to identify and implement missing functions within a codebase.

For the prototype developed in this thesis, 'pycparser' was utilized to generate the AST, identify missing functions, and automate the insertion of generated code into the existing C codebase. This library was instrumental in achieving the goal of automating the HAL generation process, ensuring that the system could iteratively analyze, modify, and extend the code with minimal manual intervention. For detailed code implementation, see the figure **??** in the appendix.

In our implementation, we generate functions that handle GPIO operations directly through register manipulation, bypassing the standard HAL functions provided by the platform. These functions are optimized for specific hardware needs and include:

- `ENABLE_GPIOA_CLK()` – Enables the clock for GPIOA.
- `set_io_mode()` – Configures a GPIO pin as either input or output.
- `HAL_gpio_write()` – Writes a value to a specific GPIO pin.
- `HAL_gpio_read()` – Reads the state of a specific GPIO pin.
- `HAL_gpio_toggle()` – Toggles the state of a GPIO pin.

To demonstrate our approach, we provide a simplified version of the `set_io_mode()` function, which directly manipulates the GPIO registers, avoiding the use of any generic HAL functions:

Listing 1: Simplified `set_io_mode()` Function

```
void set_io_mode(uint32_t gpio_base, uint32_t
    pin_mask, uint8_t mode) {
    volatile uint32_t *GPIO_MODER = (uint32_t *)(
        gpio_base + 0x00);
    uint8_t pin_number = 0;
    while ((pin_mask >> pin_number) != 1) {
        pin_number++;
    }
    *GPIO_MODER &= ~(0x3 << (pin_number * 2));
    *GPIO_MODER |= (mode << (pin_number * 2));
}
```

For further details, the complete code listings are available in the project repository[7].

3) Automated Generation of Missing Code Elements: The first goal is to test whether missing functions and code elements can be identified and generated automatically without human intervention.

In the process of generating a complete code base from an initially incomplete one, we begin by analyzing the existing structure of the program using an Abstract Syntax Tree (AST). The AST enables a deep inspection of the code, helping to identify any missing pieces by mapping the structure and the relationships within the code. Once the missing piece in the code are identified, we apply Retrieval-Augmented Generation (RAG) to automatically generate the missing code segments. This ensures that each missing piece is generated contextually based on the rest of the codebase. The result is a fully complete and functional code base, with all missing components generated seamlessly through this method.

This process is illustrated in the following figure (see Figure **??**), which demonstrates the workflow from code analysis through the AST to code completion using RAG.

[7]Repository URL or reference

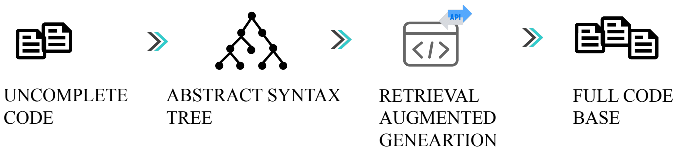


Fig. 4: **Code Generation Process.** Starting with an incomplete code base, we use an Abstract Syntax Tree (AST) to analyze the program's structure and identify the missing code pieces. Retrieval-Augmented Generation (RAG) is then used to generate each missing piece, resulting in a complete code base.

After generating the code, we will validate it with the process described below. Please refer to the diagram shown in the figure 5 for the details of the code validation process.

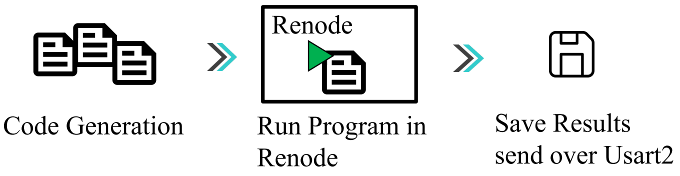


Fig. 5: **Code Validation Process.** After generating the code, the build version is run in a Renode environment. The application logs are sent over USART2 and saved for further analysis.

IV. Experimental Evaluation

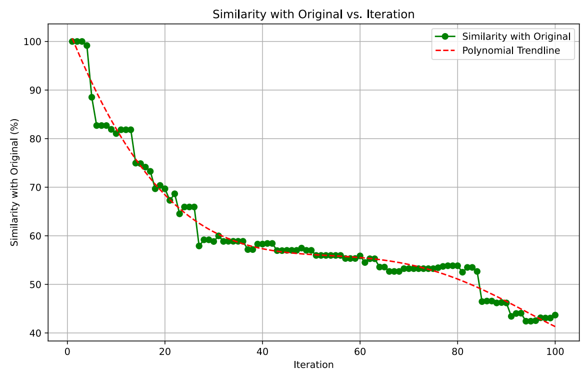


Fig. 6: **Overall Similarity Change.** The figure shows the overall similarity between the regenerated code and the initial code file across all iterations.

a) Interpretation: The results, as presented in Figures **??** and 6, illustrate the behavior of the system during the test. The similarity patterns shown in Figure 6 suggest that while the system (ChatGPT) generates code that is largely similar to the original, it is not fully deterministic in its code generation process. Despite this variability, the code was compiled successfully in every iteration, demonstrating that the system consistently generates functioning code, even as it introduces variations.

V. Discussion/Interpretation

The results Showcase the ability of LLMs to Create large Scale Software If it is clearly structered.

Although ChatGPT is Not deterministic, as can bei seen in the similarity figure, ChatGPT nonetheless creates accurate results Most of the time.

That means, If the Task ist clearly defined IT can Create the exact function and fit IT into a larger Software without Human Intervention.

VI. Threats to Validity

A threat to internal validity is the inherent non-deterministic behavior of LLMs resulting in non-repeatable probabilistic generation of prgoram code.

The construct validity describes...

The external validity refers to...

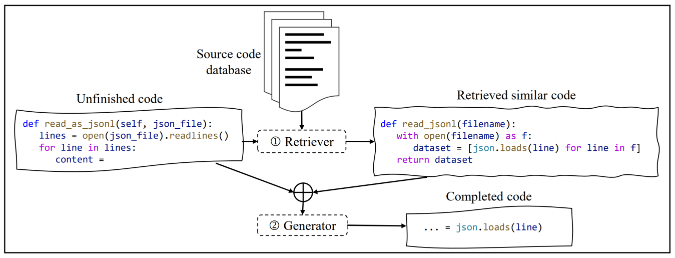
- ChatGPT 4 o Mini. API. Does evolve and maybe Adjustments have to be Made... so the prototypes ability can vary, while the underlying concept proof remains

VII. Related Work

On the machine learning side, TinyML is a growing field where small models are deployed on microcontrollers, presenting unique challenges for code generation [11]. Recent approaches like the "Outline, Then Details" framework show promise in syntactically guided code generation for embedded environments [12]. CoCoMIC introduces a new way to perform code completion by jointly modeling in-file and cross-file contexts, relevant to filling missing software components in complex embedded systems [13].

LLM-based approaches are also being explored for self-planning in code generation, where large language models can generate code autonomously based on a structured plan [14]. Techniques like Retrieval-Augmented Code Generation (RAG) are becoming more prevalent, as they can efficiently generate missing software components by retrieving relevant examples from existing codebases [15]. LLM-based control code generation solutions are gaining traction in industrial settings for generating real-time control code for embedded systems [16].

ReACC is a retrieval-augmented code completion framework that retrieves similar code snippets from a database to use as external input for code completion [17]. The ReACC framework (see Figure 7) enhances code completion by incorporating retrieved code examples, which is relevant to our approach of using context from existing codebases.



```
def read_as_jsonl(self, json_file):
   lines = open(json_file).readlines()
   for line in lines:
      content =
```

```
def read_jsonl(filename):
   with open(filename) as f:
      dataset = [json.loads(line) for line in f]
   return dataset
```

```
... = json.loads(line)
```

Fig. 7: **ReACC Framework Illustration [17].** This figure illustrates the ReACC framework, which retrieves similar code snippets from a database to use as external input for code completion.

Lastly, the impact of code language models on automated program repair has been studied, providing insights into how these models can assist in creating robust, error-free software components for embedded systems [18]. Future work can build on these approaches by integrating hybrid techniques like GNNs for more refined and context-aware code generation [19].

VIII. Outlook

A. Conclusion

The proposed method demonstrates significant potential for streamlining the code generation process, particularly in the domain of embedded systems. By automating the identification and generation of hardware-specific code, this approach minimizes the need for developers to manually look up microcontroller-specific details or select appropriate libraries. As a result, the code generation process becomes more adaptable across different microcontroller platforms, offering a flexible and efficient solution.

Based on expert feedback, future iterations of this prototype should focus on incorporating hardware addresses from the CMSIS layer. This enhancement would not only align the prototype more closely with industry standards but also effectively segregate hardware-specific code into distinct layers, addressing concerns regarding maintainability and scalability in more complex systems.

In light of these insights, future work will explore the following hypotheses to guide further development and improvements in the system.

B. Future Work

Further research is necessary to improve the system's capacity to handle increasingly complex and varied programming environments, as well as to integrate support for additional microcontroller architectures.

a) Hypothesis 1: Integration of Datasheet Information: Can the integration of datasheet information into the code generation process improve the applicability of the prototype across different microcontroller families?

b) Hypothesis 2: Enhancing Hardware Abstraction: Can the code generation process, with its AST-based implementation, be extended to function effectively across a broader range of microcontroller families?

Acknowledgment

This work was supported by AGSOTEC.

Glossary

AST: Abstract Syntax Tree A tree representation of the abstract syntactic structure of source code written in a programming language.

GPT-4o Mini: Large Language Model from OpenAI A compact variant of the GPT-4 language model designed for cost-efficient and versatile tasks.

HAL: Hardware Abstraction Layer A layer of programming that allows a computer operating system to interact with a hardware device at an abstract level.

LLM: Large Language Model A type of artificial intelligence designed to understand and generate human-like text.

RAG: Retrieval-Augmented Generation A process that enhances large language models by allowing them to respond to prompts using a specified set of documents.

STM32F407: High-performance Microcontroller A microcontroller that offers the performance of the Cortex-M4 core.

References

[1] J. Justyna. "Global Surge in Embedded Software Demand: Here is Why". In: *DAC.digital* (Dec. 2023). [Online]. Accessed: Aug. 29, 2024. URL: https://dac.digital/global-surge-in-embedded-software-demand-here-is-why/ (cited on page 1).

[2] J. Beningo. *Embedded Basics – API's vs HAL's*. Accessed: 2024-08-21. BeningoEmbeddedGroup, Apr. 2016. URL: https://www.beningo.com/embedded-basics-apis-vs-hals/ (cited on page 1).

[3] Y. Zhu. *Embedded Systems with Arm Cortex-M Microcontrollers in Assembly Language and C*. English. 2nd. Lulu, NC: E-MAN PR LLC, 2015. ISBN: 0982692633 (cited on page 2).

[4] T. Mikolov, K. Chen, G. Corrado, and J. Dean. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL]. URL: https://arxiv.org/abs/1301.3781 (cited on page 2).

[5] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou. *The Faiss library*. 2024. arXiv: 2401.08281 [cs.LG]. URL: https://arxiv.org/abs/2401.08281 (cited on page 2).

[6] STMicroelectronics. *STM32F4DISCOVERY*. (n.d.). [Online]. Accessed: 2024-08-29. URL: https://www.st.com/en/evaluation-tools/stm32f4discovery.html (cited on page 2).

[7] E. Urban. *Introduction to Prompt Engineering. Microsoft Learn*. Accessed: 2024-03-29. 2024. URL: https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/prompt-engineering (cited on page 3).

[8] E. Urban. "Prompt Engineering Techniques". In: *MicrosoftLearn* (2024). URL: https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/advanced-prompt-engineering (visited on 02/16/2024) (cited on page 3).

[9] deepset Cloud. *Prompt Engineering Guidelines*. Accessed: 2024-04-01. Apr. 2024. URL: https://docs.cloud.deepset.ai/docs/prompt-engineering-guidelines (cited on page 3).

[10] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd. Upper Saddle River, NJ: Pearson Education, Inc, 2006. ISBN: 0-201-10088-6 (cited on page 3).

[11] I. Siqueira. "TinyML: The Future of Embedded Machine Learning is to Change Lives for the Better". In: *Medium* (Apr. 2021). URL: https://towardsdatascience.com/tinyml-the-future-of-embedded-machine-learning-is-to-change-lives-for-the-better-87230668a08c (cited on page 5).

[12] W. Zheng, S. P. Sharan, A. K. Jaiswal, K. Wang, Y. Xi, D. Xu, and Z. Wang. *Outline, Then Details: Syntactically Guided Coarse-To-Fine Code Generation*. Accessed: 2024-10-01. 2023. arXiv: 2305.00909 [cs.CL]. URL: https://arxiv.org/abs/2305.00909 (cited on page 5).

[13] Y. Ding, Z. Wang, W. U. Ahmad, M. K. Ramanathan, R. Nallapati, P. Bhatia, D. Roth, and B. Xiang. "CoCoMIC: Code Completion By Jointly Modeling In-file and Cross-file Context". In: (2023). [Online]. Available: https://arxiv.org/abs/2212.10007. Accessed: 2024-10-01 (cited on page 5).

[14] X. Jiang, Y. Dong, L. Wang, Z. Fang, Q. Shang, G. Li, Z. Jin, and W. Jiao. *Self-planning Code Generation with Large Language Models*. Accessed: 2024-10-01. 2024. arXiv: 2303.06689 [cs.CL]. URL: https://arxiv.org/abs/2303.06689 (cited on page 5).

[15] M. R. Parvez, W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang. "Retrieval-Augmented Code Generation and Summarization". In: (2021). [Online]. Available: https://arxiv.org/abs/2108.11601. Accessed: 2024-10-01 (cited on page 5).

[16] H. Koziolek, S. Grüner, R. Hark, V. Ashiwal, S. Linsbauer, and N. Eskandani. "LLM-based and Retrieval-Augmented Control Code Generation". In: *LLM4Code 2024*. ABB Research, Germany. Apr. 2024 (cited on page 5).

[17] S. Lu, N. Duan, H. Han, D. Guo, S.-w. Hwang, and A. Svyatkovskiy. *ReACC: A Retrieval-Augmented Code*

Completion Framework. Accessed: 2024-10-01. 2022. arXiv: 2203.07722 [cs.CL]. URL: https://arxiv.org/abs/2203.07722 (cited on page 5).

[18] N. Jiang, K. Liu, T. Lutellier, and L. Tan. “Impact of Code Language Models on Automated Program Repair”. In: (2023). [Online]. Available: https://arxiv.org/abs/2302.05020. Accessed: 2024-10-01 (cited on page 5).

[19] S. Liu, Y. Chen, X. Xie, J. Siow, and Y. Liu. “Retrieval-Augmented Generation for Code Summarization via Hybrid GNN”. In: (2021). [Online]. Available: https://arxiv.org/abs/2006.05405. Accessed: 2024-10-01 (cited on page 5).