**T.C.**

**İSTANBUL KÜLTÜR UNIVERSITY**

**FACULTY OF ENGINEERING**

**DEPARTMENT OF**

**ELECTRICAL & ELCTRONICS ENGINEERING**

|  |
| --- |
| IKULogo **CODE GENERATION IN EMBEDDED SYSTEMS VIA LARGE LANGUAGE MODELS (LLMS)** |

**Graduation Project**

**Name SURNAME: DENİZ METE COŞKUNSU**

**2000003416**

**Name SURNAME**

**Lecturer BASRİ ERDOĞAN**

**Supervisor**

**JUNE 2025**

ACKNOWLEDGEMENT

I would like to express my heartfelt gratitude to everyone who supported and contributed to the completion of this project. First of all, I would like to express my deepest gratitude to my advisor Basri Erdoğan. I would also like to express my gratitude to our instructors, Mert Hayar and Ahmet Aytuğ Ayrancı, who guided us throughout the project. Finally, I would like to thank my family and friends for always making us feel their support.

ABSTRACT

Embedded systems play a vital role in domains such as automotive electronics, industrial automation, and healthcare devices. Developing reliable embedded system software often requires generating optimized C code, making the process both time-consuming and error-prone. This study presents a fine-tuned language model specifically designed to generate embedded C code locally, securely, and efficiently.

A custom, domain-specific dataset was constructed by mining and filtering real-world embedded C code examples from open-source GitHub repositories. Using a Python-based data collection filter pipeline, 19,769 unique code samples totaling 69,785 lines were obtained. The dataset was refined using keyword- and file-based filtering to ensure that only high-quality, embedded-relevant code was included.

StarCoderBase-1B is selected as the model due to its 1 billion parameters and code generation performance. Fine-tuning (fine-tuning) is performed using a single A100 GPU on Google Colab Pro+ with the QLoRA method and 4-bit quantization technology. The entire training process, from the payment of the dataset to tokenization and model adaptation, is supervised to check resource efficiency and repeatability..

Evaluation results indicate that the fine-tuned model achieved low training and validation loss values (0.6001 and 0.5441, respectively) with stable convergence. The generated outputs were syntactically correct and well-aligned with embedded coding patterns. This project demonstrates that with carefully curated data and efficient training techniques, large language models can become effective, privacy-preserving assistants in embedded software development.

ÖZET

Gömülü sistemler; otomotiv elektroniği, endüstriyel otomasyon ve sağlık teknolojileri gibi birçok alanda kritik rol oynamaktadır. Güvenilir gömülü yazılım geliştirmek genellikle optimize edilmiş C kodlarının üretilmesini gerektirir ve bu süreç oldukça zaman alıcı ve hata yapmaya açıktır. Bu çalışma, gömülü C kodunu yerel, güvenli ve verimli bir şekilde üretebilen, özel olarak eğitilmiş bir dil modeli sunmaktadır.

Çalışmada, açık kaynaklı GitHub depolarından gerçek gömülü C kodu örnekleri anahtar kelime ve dosya adı tabanlı filtrelerle toplanarak özgün ve alanına özel bir veri seti oluşturulmuştur. Python tabanlı veri toplama pipeline’ı ile 19.769 kod örneği ve toplamda 69.785 satır kod elde edilmiştir. Veri seti, yalnızca gömülü sistemlerle doğrudan ilişkili yüksek kaliteli kodların dahil edilmesini sağlamak amacıyla anahtar kelime ve dosya filtrelemesi ile oluşturulmuştur.

Model olarak, 1 milyar parametreye sahip olması ve kod üretim performansı sayesinde StarCoderBase-1B seçilmiştir. İnce ayar (fine-tuning), QLoRA yöntemi ve 4-bit quantization tekniği ile Google Colab Pro+ üzerinde tek bir A100 GPU kullanılarak gerçekleştirilmiştir. Veri setinin toplanmasından tokenizasyon ve model adaptasyonuna kadar tüm eğitim süreci, kaynak verimliliği ve tekrarlanabilirlik gözetilerek uygulanmıştır.

Değerlendirme sonuçları, ince ayar yapılan modelin düşük eğitim ve doğrulama loss değerlerine ulaştığını (sırasıyla 0.6001 ve 0.5441) ve istikrarlı bir şekilde yakınsadığını göstermektedir. Üretilen çıktılar, sentaks açısından doğru ve gömülü kodlama desenleriyle uyumludur. Bu proje, dikkatli veri seçimi ve verimli eğitim teknikleri ile büyük dil modellerinin gömülü yazılım geliştirmede etkili ve mahremiyete duyarlı asistanlar haline gelebileceğini göstermektedir.

TABLE OF CONTENTS

ACKNOWLEDGEMENT ii

ABSTRACT iii

ÖZET iv

LIST OF FIGURES vii

LIST OF TABLES viii

1. INTRODUCTION 1

1.1. Importance of AI Assistant in Embedded Systems 1

1.2. Literature Review and Motivation 2

1.3. Key Contributions of the Research 3

1.3.1. Local and Privacy-Friendly Operation 3

1.3.2. Domain-Specific Dataset and Customized Filtering 4

1.3.3. Efficient Fine-tuning with QLoRA 4

1.3.4. Enhanced Code Quality and Developer Efficiency 4

1.4. Structured Methodology for Embedded LLM Fine-Tuning 4

1.4.1. Data Collection and Pre-processing 5

1.4.2. Code Refinement and Tokenization Pipeline 5

1.4.3. LLM Model Selection and Training 5

1.4.4. Output Evaluation and System Validation 6

1.5. Project Impact 6

2. METHODS 8

2.1. Data Collection 8

2.2. Data Filtering Approach 8

2.3. Data Representation and Tokenization 9

2.3.1. Tokenization Strategy 10

2.4. Model Selection and Training 10

2.4.1. Fine-Tuning Configuration Details 11

2.4.2. Training Strategy and Optimization 12

3. IMPLEMENTATION AND RESULTS 15

3.1. Dataset Acquisition and Cleaning 15

3.1.1. Filtering and Preprocessing 15

3.1.2. Tokenization Pipeline 16

3.2. Model Architecture 17

3.3. Training Environment Setup 18

3.4. Fine-Tuning with QLoRA 19

3.4.1. Quantization Configuration 19

3.4.2. LoRA Adapter Configuration 19

3.4.3. Training Arguments and Strategy 20

3.4.4. Callbacks and Monitoring 21

3.5. Evaluation Metrics and Results 21

3.5.1. Training and Validation Loss 22

3.5.2. Prompt-Based Evaluation of Embedded Peripheral Scenarios 22

3.5.3. Comparative Analysis of Base and Fine-Tuned Model Outputs 23

3.6. Challenges Encountered During Implementation 25

4. CONCLUSION 26

4.1. Future Works 27

5. REFERENCES 29

6. APPENDIX A 30

7. CV: 31

# LIST OF FIGURES

Figure 1 22

Figure 2 24

Figure 3 24

# LIST OF TABLES

[Table 1 18](#_Toc200620067)

# INTRODUCTION

Embedded systems are special-purpose systems that can interact with the physical world through microcontrollers and software built into the hardware. In electrical and electronics engineering, they are used in many important areas such as automotive control units, industrial sensors, medical devices, and IoT-based monitoring systems. These systems usually use C or C++ programming languages because they need to work with limited hardware resources and real-time timing requirements.

However, developing embedded software is a difficult task. It often includes working directly with hardware registers, using low-level APIs, and writing code for real-time operating systems (RTOS). These steps take a lot of time and are easy to make mistakes in.

In recent years, Large Language Models (LLMs) have shown strong performance in general code generation. But these models are trained with a wide range of programming data, and they are not very good at specific technical areas like embedded systems.

This study introduces a small and customized LLM that focuses on generating embedded C code. The goal is to build an AI assistant that works locally and can create syntactically correct and hardware-friendly code. For this purpose, a special dataset was created by filtering and cleaning real embedded system projects from GitHub. Then, the model was fine-tuned using the QLoRA method.

The rest of this thesis explains the importance of AI assistants in embedded systems, reviews similar studies, and describes the technical steps followed in this project.

## ****Importance of**** AI Assistant in Embedded Systems

Embedded system developers often have to deal with complex tasks that require a lot of attention, experience, and careful planning. Their daily tasks include setting up microcontrollers, managing GPIO pins, managing interrupts, and configuring timers. These tasks must be done correctly to comply with both coding conventions and hardware requirements.

Since these tasks are quite demanding, developers often find themselves rewriting similar pieces of code over and over, constantly checking hardware manuals, and spending extra time looking for errors during integration. These processes slow down the overall development pace.

With Large Language Models (LLMs) rapidly improving, having an AI assistant that can generate custom embedded system code can be very useful. Such an AI that works completely offline on local devices also helps avoid security issues associated with cloud services. Such a system can help developers minimize errors, keep their code consistent, and make learning embedded systems easier for beginners.

## ****Literature Review and Motivation****

Large language models (LLMs) have made significant progress in software development processes, especially in the area of ​​code generation. The Codex model developed by OpenAI was trained on billions of lines of GitHub code by fine-tuning GPT-3 and was shown to be able to produce functionally correct code on a large number of programming tasks (Chen et al., 2021). Following this, models such as StarCoder (Li et al., 2023) and Code Llama achieved higher success on similar tasks by pre-training on larger code datasets and fine-tuning with data specific to specific programming languages.

In order to obtain more secure outputs in code generation, some studies recommend fine-tuning, especially with commits that fix vulnerabilities. For example, Li et al. (2024) increased secure code generation by 5–6% by training models with more than 14,000 C/C++ code fixes. Similarly, Pearce et al. (2022) showed that GitHub Copilot produced C codes with 40% security vulnerabilities when driven by dangerous functions. This situation reveals that LLMs are directly related to the quality of the data on which they are trained.

On the other hand, in hardware-focused tests conducted by Englhardt et al. (2023), it was shown that models such as GPT-4 can produce fully functional codes at the microcontroller level and at the same time explain the solution logic in natural language. Such explanations are beneficial not only in terms of code generation but also in terms of guiding the developer. However, the success of the models varies according to the complexity of the task and in some cases they are insufficient without human supervision.

A similar approach was adopted in this dissertation; a domain-specific C/C++ dataset for embedded systems was constructed by directly mining open-source GitHub repositories. Using a custom Python-based data collection pipeline, code files containing embedded-related keywords in their names (such as "gpio", "stm32", "spi", "i2c", etc.) were systematically gathered. To ensure high data quality, additional filters were applied to include only sufficiently long and relevant code samples while eliminating duplicates through hash checking. The resulting dataset-composed of 19,769 unique embedded C code examples- was then used to fine-tune the StarCoderBase-1B model using the QLoRA method. This pipeline allowed for precise domain alignment and robust code generation performance tailored to embedded software development.

The academic sources cited above provided both technical ground and methodological guidance while establishing the infrastructure of this study. These references directly contributed to the design of the system and formed the background in shaping the dataset used. Thus, this thesis study practically implemented the goal of generating LLM-based secure and descriptive code for embedded systems in line with the existing literature.

## Key Contributions of the Research

The main goal of this project is to provide an AI assistant that can generate embedded C code designed for microcontroller applications. While general-purpose code generation by large language models (LLMs) has been widely studied, high-quality code generation for embedded systems remains challenging and less researched. This thesis addresses these gaps with the following unique contributions:

### Local and Privacy-Friendly Operation

Unlike many current code-generation systems, this AI assistant runs entirely on local devices. This avoids potential cloud-related security risks and protects proprietary or sensitive embedded system code. Additionally, local execution provides a faster and more accessible user experience.

### Domain-Specific Dataset and Customized Filtering

The project uses a domain-specific dataset created by systematically collecting embedded C code directly from open-source GitHub repositories. A custom data collection pipeline was implemented to collect github project files whose names contained embedded-related keywords (e.g., “gpio”, “stm32”, “spi”, “i2c”, etc.). To maximize data quality and domain relevance, the pipeline included file-based and keyword-based filtering, minimum length checks, and duplicate elimination via hash validation. This targeted approach directly increased the effectiveness of the fine-tuned model by ensuring that only high-quality, embedded-specific code was included in the training data.

### Efficient Fine-tuning with QLoRA

To efficiently train the model, the QLoRA technique was chosen, enabling the fine-tuning of the StarCoderBase-1B model with limited computational resources. The use of 4-bit quantization significantly reduced the memory footprint while maintaining excellent performance in code generation tasks.

### Enhanced Code Quality and Developer Efficiency

This AI assistant aims to significantly speed up the embedded software development process. It generates syntactically correct and hardware-compatible code snippets, reducing manual errors, repetitive coding, and dependency on hardware documentation.

These contributions collectively represent an innovative approach to integrating LLMs within embedded systems development, offering practical, secure, and efficient solutions customised for the embedded software industry.

## Structured Methodology for Embedded LLM Fine-Tuning

This project uses a systematic and structured approach to develop a local AI assistant designed for embedded C code generation. The methodology is summarized below:

### Data Collection and Pre-processing

In this study, the dataset was constructed by systematically collecting embedded C code directly from open-source GitHub repositories. Using a custom Python-based pipeline, only .c files whose filenames included embedded-relevant keywords were selected. To ensure data quality and relevance, samples were filtered based on minimum length and checked for duplicates via hash matching.

### Code Refinement and Tokenization Pipeline

* Files containing embedded-specific keywords were included and irrelevant files were discarded.
* License comments and legal information were removed to prevent noise during training.
* The curated code samples were stored in .jsonl format, making them compatible with downstream tokenization tools.
* StarCoderBase-1B tokenizer was used to tokenize the dataset in preparation for model fine-tuning.

### LLM Model Selection and Training

* StarCoderBase-1B was selected due to its number of parameters, ability to generate code, good performance, and ability to run on local hardware.
* The QLoRA method was used for fine-tuning, which allowed efficient training with low memory usage through 4-bit quantization.

#### Training Configuration Summary:

* **Model:** StarCoderBase-1B
* **Fine-Tuning Method:** QLoRA
* **Quantization:** 4-bit (NF4) with bfloat16 compute
* **LoRA Target Modules:** ["c\_attn", "c\_proj", "c\_fc", "c\_mlp"]
* **LoRA Rank (r):** 8
* **LoRA Alpha:** 16
* **LoRA Dropout:** 0.05
* **Epochs:** 2
* **Total Training Samples:** 69,785
* **Batch Size:** 6
* **Optimizer:** paged\_adamw\_8bit
* **Learning Rate Scheduler:** Cosine
* **Checkpointing:** Every epoch
* **Callbacks:**

1. PrintLossCallback (for clear loss reporting at the end of each epoch)

### Output Evaluation and System Validation

The model’s performance was tracked using training and validation loss metrics, and the generated embedded C code was tested for both syntax correctness and hardware-level compatibility, ensuring the development of a secure, efficient, and task-specific AI assistant for embedded systems, with detailed steps and code samples explained in the following sections.

## ****Project**** Impact

Embedded systems are used in critical areas such as automotive control, medical devices, and industrial automation. These systems often require optimized and error-free code, which makes the development process difficult, time-consuming, and open to human mistakes. This project presents a local AI assistant that can support developers by generating embedded C code, improving both development speed and code quality.

A key feature of this work is that the model runs offline, on local hardware. This provides strong privacy and removes the risks of sharing sensitive code with cloud servers. This is especially useful in secure areas such as defense technologies or medical equipment. Also, by using a small and efficient model like StarCoderBase-1B and training it with QLoRA, we were able to perform fine-tuning on limited hardware without losing performance.

Another major contribution is the use of a domain-specific dataset, filtered to include only relevant embedded code. This helps the model generate more accurate and useful results for embedded software tasks. As a result, even beginner developers can create working code more easily and learn faster without relying heavily on technical documents.

In summary, this thesis is a step towards developing smarter and more secure software for embedded systems. The methods and tools developed here can make embedded programming faster, safer, and easier to learn.

# METHODS

This section outlines the methodological framework used to develop a lightweight, domain-specific language model for embedded C code generation. The workflow includes data acquisition, filtering strategies, data representation, model selection rationale, and fine-tuning methodology. Rather than focusing on implementation details, this chapter aims to describe the reasoning and structure behind the techniques applied in this research.

## Data Collection

In machine learning applications for code generation, the quality and relevance of the training data play a crucial role in the model’s final performance. This project required a real-world dataset that accurately reflects embedded system development practices. To address this, a custom dataset of embedded C code was constructed directly from open-source GitHub repositories.

A Python-based data collection pipeline was developed, systematically searching for repositories using queries related to embedded systems (such as "stm32", "avr", "pic", "gpio", etc.). For each repository, only .c files whose filenames contained embedded-related keywords were considered. To further ensure the quality of the dataset, code samples were included only if they contained at least 20 lines of code and basic C-style comments (to ensure source code, not binary or data dumps).

Duplicate samples were eliminated using hash checks, ensuring each code block in the dataset was suitability. In total, 19,769 unique embedded C code examples comprising 69,785 lines were collected and stored in .jsonl format for further processing.

## Data Filtering Approach

To maximize dataset relevance and quality, a structured multi-stage filtering approach was implemented during data collection. The key filtering steps are outlined below:

**Keyword-Driven Repository and File Selection**

The initial stage of filtering involved querying the GitHub API with multiple search terms related to embedded systems (e.g., "stm32", "avr", "pic", "gpio", "adc", "i2c", "spi", "hal", "driver"). Within each repository, only files with a .c extension and filenames containing embedded-specific keywords were selected. This ensured the initial pool of code samples was highly likely to contain embedded C code.

**Minimum Content and Structure Checks**

Selected files were required to contain at least 20 lines of code to exclude trivial, incomplete, or non-source files. Additionally, basic syntactic checks—such as the presence of C-style comments (// or /\* \*/)—were applied to further filter out non-informative or autogenerated content.

**Duplicate Removal**

To ensure dataset uniqueness, an MD5 hash was computed for the contents of each code file. Files producing the same hash were identified as duplicates and only a single instance was retained. This process eliminated redundant training examples and maximized the diversity of code structures.

**License and Legal Statement Removal**

Legal and copyright-related comment blocks were programmatically removed using string pattern matching and regular expressions. This step prevented legal boilerplate from biasing the model and ensured only functional code patterns contributed to learning.

**Output Formatting**

After filtering, all code samples were saved in the .jsonl (JSON Lines) format, with each line representing a unique code block. This structure facilitated downstream tokenization and model training.

Through this multi-stage filtering pipeline, the final dataset was curated to ensure high quality, domain-specificity, and diversity—critical factors for effective language model fine-tuning in embedded systems development.

## Data Representation and Tokenization

To prepare the cleaned dataset for fine-tuning, each code block was first serialized into a machine-readable format. The .jsonl (JSON Lines) format was chosen, where each line contains a single JSON object, the “code” field. Converting the data to JSONL for the tokenizer process allows processing each text sample separately and in a standardized structure to train the model efficiently and without errors.

{"code": "void main() {\n HAL\_GPIO\_TogglePin(GPIOA, GPIO\_PIN\_5);\n}"}

### Tokenization Strategy

The next step was transforming this structured textual data into token sequences suitable for language modeling. For this purpose, the StarCoderBase-1B tokenizer, developed as part of the BigCode initiative, was employed. This tokenizer is optimized for source code and supports a wide range of programming languages, including embedded C/C++.

Key aspects of the tokenization process included:

* **Tokenizer Type:** AutoTokenizer from HuggingFace with use\_fast=True enabled to improve tokenization speed.
* **Padding Strategy:** The eos\_token was assigned as the pad\_token to maintain consistent sequence lengths across batches.
* **Max Length:** Sequences were truncated or batched to a maximum of 512 tokens for memory-efficient training.
* **Batch Streaming:** Tokenization was applied in streaming batches to avoid GPU/CPU memory overflow, allowing the process to scale up to millions of examples.
* **Storage Format:** Tokenized data was saved using Python's pickle module as .pkl files for fast I/O during training.

## Model Selection and Training

This study employed the StarCoderBase-1B model, an open-source large language model (LLM) developed by the BigCode initiative. The model was specifically chosen for its balance between computational efficiency and strong performance on code generation tasks.

StarCoderBase-1B was pretrained on a diverse multilingual code dataset (The Stack v1.2) containing source code from over 80 programming languages. With a 1 billion parameter architecture and a context window of 8192 tokens, it supports efficient handling of long code sequences and structured syntax. The model’s Fill-in-the-Middle (FIM) training objective allows it to predict missing code segments when given both preceding and following context, making it particularly effective for code completion, refactoring, and generation.

To make the training process feasible within limited hardware constraints, a parameter-efficient fine-tuning (PEFT) strategy was adopted. Specifically, the QLoRA (Quantized Low-Rank Adaptation) technique was used, which enables fine-tuning of large language models by combining:

* 4-bit quantization, which compresses model weights to reduce memory usage during training, and
* Low-Rank Adaptation (LoRA), which injects trainable rank-decomposed matrices into specific attention and feedforward layers, allowing adaptation without updating the full set of model parameters.

This hybrid method significantly reduces memory and computation costs while maintaining model accuracy. The training process focused on adapting the model to embedded C code patterns using domain-specific data prepared in earlier stages. Hyperparameter selection, optimization strategy, and validation procedures were applied with the aim of maximizing convergence speed while avoiding overfitting.

The overall training workflow was designed to maintain reproducibility and efficiency, using established open-source frameworks and accessible hardware resources.

### Fine-Tuning Configuration Details

To fine-tune the StarCoderBase-1B model for embedded C code generation, the QLoRA method was employed, which combines 4-bit quantization with LoRA (Low-Rank Adaptation). This hybrid approach enables efficient adaptation of large language models using significantly fewer trainable parameters.

#### LoRA Mathematical Configuration

In a standard transformer layer, linear projections are represented as:

Where:

* X is the input vector,
* is the full-rank weight matrix,
* **y** is the output of the linear layer.

With LoRA, instead of updating the full matrix **W**, a low-rank update is introduced:

Where:

* and are trainable low-rank matrices,
* r is the **rank** of the decomposition (in study, r = 16),
* α is the **scaling factor** to stabilize training (set to α=32),
* represents the low-rank parameter delta added to **W**.

This formulation reduces the number of trainable parameters by fine-tuning only A and B and preserves the model’s capacity to adapt to new tasks by leveraging low‐rank adaptations to approximate full‐rank updates with minimal performance loss (Hu et al., 2021).

A dropout value of 0.05 was applied to the LoRA modules to mitigate overfitting during training.

#### Quantization Setup

Quantization was applied using the BitsAndBytes library, converting weights to 4-bit NF4 format. Forward and backward computations were carried out using **bfloat16** (BF16) precision to minimize memory usage while preserving numerical stability.

This configuration enabled efficient training on a single A100 GPU with approximately 0.8 GB memory overhead for trainable parameters.

### Training Strategy and Optimization

The fine-tuning process of the StarCoderBase-1B model was carried out using the Hugging Face Trainer API, which provides a robust and reproducible training pipeline for large language models. The goal was to adapt the pretrained model to domain-specific patterns in embedded C code, while ensuring efficient convergence and minimal overfitting.

#### Training Arguments and Hyperparameters

The training configuration was carefully selected to balance computational resource constraints and model performance. Based on empirical testing and memory limitations, the following hyperparameters and settings were adopted:

* **Epochs:** 2

Training was performed for two full epochs on the entire dataset of 19,769 code examples, enabling effective domain adaptation while mitigating the risk of overfitting.

* **Batch Size:** 6

To accommodate GPU memory constraints, a per-device batch size of 6 was used. Gradient accumulation with 2 steps was enabled, resulting in an effective batch size of 12.

* **Learning Rate:**

Tuned to avoid divergence or slow convergence.

* **Precision:** bfloat16 (BF16)

Reduced memory usage while maintaining numerical stability on A100 GPU.

* **Optimizer:** paged\_adamw\_8bit

Memory-efficient optimizer suitable for quantized models. Offered better scalability during large-scale training.

* **Scheduler:** Cosine

Helped in decaying the learning rate smoothly over time, which often results in improved convergence.

* **Logging & Checkpointing:**

Training logs were saved every 50 steps for real-time monitoring.

Model checkpoints were written to Google Drive every epoch to ensure fault tolerance.

* **Custom Callback:**

A custom PrintLossCallback was implemented to display clear train and evaluation loss values at the end of each epoch, enhancing training transparency.

* **Additional Settings:**

gradient\_checkpointing was enabled to further optimize memory usage during training.

The model was configured to automatically load the best-performing checkpoint (lowest validation loss) at the end of training.

These settings ensured that the training process was both resource-efficient and robust, making it feasible to fine-tune a large language model for embedded code generation on standard cloud GPU resources.

# IMPLEMENTATION AND RESULTS

This section details the practical implementation steps of the proposed approach, including dataset preparation, model configuration, training setup, and observed outcomes. Each phase is described with the specific tools, techniques, and configurations used during development.



## Dataset Acquisition and Cleaning

In this study, the training dataset was constructed by directly collecting embedded C code from open-source GitHub repositories. A custom Python pipeline automated the search and download process by using targeted queries focused on embedded systems topics such as STM32, AVR, PIC, GPIO, I2C, SPI, and HAL. Only .c files whose filenames contained embedded-relevant keywords were included in the dataset.

To ensure high data quality, each file was required to have at least 20 lines of code and to contain C-style comments. This filtering process helped eliminate incomplete, trivial, or non-informative files. Duplicate code blocks were identified using MD5 hashing, and only unique samples were retained. License texts and legal statements were programmatically removed, ensuring that the dataset consisted solely of functional embedded system code.

The final dataset consisted of 19,769 unique embedded C code samples, representing a total of 69,785 lines. These samples span a wide variety of microcontroller applications, including peripheral control, hardware drivers, interrupt handling, and real-time routines. All collected data was stored in .jsonl format for efficient downstream processing and model fine-tuning.

### Filtering and Preprocessing

To filter and preprocess the dataset for embedded-specific code, a custom Python pipeline was implemented.

The key steps are summarized below, with representative code excerpts:

**Keyword-Based File Filtering:**

Only .c files with embedded-related keywords in their filenames were processed:

def file\_is\_relevant(filename):

    keywords = ['gpio', 'spi', 'i2c', 'adc', 'uart', 'hal', 'driver', 'periph', 'stm32', 'pic', 'avr', 'msp430']

    filename\_lower = filename.lower()

    return any(k in filename\_lower for k in keywords)

**Duplicate Removal Using Hashing:**

An MD5 hash was computed for each code block. Duplicate hashes were skipped:

code\_hash = hashlib.md5(code\_content.encode()).hexdigest()

if code\_hash in hash\_set:

    return

hash\_set.add(code\_hash)

save\_checkpoints()

**License Removal:** Removed license statements and copyright headers using another regex-based function to avoid biasing the model with license text.

LICENSE\_START\_PATTERNS = [

    r"\\*{5,}", r"@file", r"@author", r"Copyright",

    r"licensed under", r"This software is licensed",

    r"AS-IS", r"All rights reserved"

]

**Format Conversion:** After cleaning and filtering, the remaining code was saved in .jsonl format (one JSON object per line), compatible with HuggingFace tokenizers.

{"code": "void main() {\n HAL\_GPIO\_TogglePin(GPIOA, GPIO\_PIN\_5);\n}"}

This structured format was essential for batching and tokenizing in the next step.

### Tokenization Pipeline

Before training the model, the cleaned dataset in .jsonl format was tokenized to convert raw text into numerical input sequences compatible with the model architecture. This step is critical to preserve the syntactic and structural integrity of embedded C code while enabling efficient GPU-based processing.

In this project, the StarCoderBase-1B tokenizer from the BigCode was used, leveraging HuggingFace's AutoTokenizer for efficient handling of source code in multiple embedded programming languages.

Key configurations and steps during tokenization included:

* The eos\_token (end-of-sequence) was set as the pad token to ensure proper termination of sequences.
* The maximum sequence length was capped at 512 tokens, suitable for memory constraints during fine-tuning.
* All tokenized outputs were stored as a list of dictionaries and serialized into a .pkl file using Python's pickle module for rapid data loading during training.
* The tokenization process was performed in a streaming batch fashion to avoid memory crashes, especially when processing large-scale datasets.

## ****Model Architecture****

For this project, StarCoderBase-1B, large language model(LLM) developed by the BigCode project, was selected. This model strikes a strong balance between computational efficiency and performance, making it highly suitable for local deployment environments typical in embedded system development.

StarCoderBase-1B consists of approximately 1 billion parameters and was pre-trained on over 80 programming languages using The Stack v1.2 dataset. It supports a context window of 8192 tokens and was trained with the Fill-in-the-Middle (FIM) objective, which enables the model to predict missing code segments given both the preceding and following context-making it particularly effective for code completion and generation tasks.

Several technical characteristics led to the selection of this model:

**Compact size:** Unlike larger models (e.g., 6B+ parameters), the 1B version can be fine-tuned on a single A100 GPU using quantized methods such as QLoRA, enabling cost-efficient training.

**Code-centric architecture:** StarCoderBase-1B was trained specifically for programming tasks and supports multi-query attention, which reduces memory consumption during inference.

**Open-source availability:** The model and tokenizer are fully open-source and actively maintained, allowing reproducibility and modification.

The following table summarizes key specifications of the pre-trained model:

|  |  |
| --- | --- |
| **Property** | **Description** |
| Model Name | StarCoderBase-1B |
| Pretrained Dataset | The Stack v1.2 (code from GitHub, 80+ langs) |
| Context Window | 8192 Tokens |
| Training Objective | Fill-in-the-Middle (FIM) |
| Parameter Count | 1 Billion |
| Attention Mechanism | Multi-Query Attention |
| Quantization Support | 4-bit NF4, bfloat16 |
| Source Repository | BigCode GitHub |

Table 1

**Pretraining Configuration:**

The base model was originally trained by the BigCode team using substantial computational resources to ensure high-quality performance. The training setup is as follows:

* **Pretraining Steps**: 500,000
* **Pretraining Tokens**: 1 Trillion
* **Precision**: bfloat16 (supports efficient mixed-precision training)
* **Hardware**: 128 × NVIDIA A100 GPUs
* **Training Time**: 11 days

These robust pretraining characteristics made StarCoderBase-1B an ideal candidate for domain-specific fine-tuning within the hardware-constrained context of this research.

## Training Environment Setup

The fine-tuning process was relized using Google Colab Pro+, equipped with an NVIDIA A100 GPU. The A100 provides 40 GB of high-bandwidth memory and supports bfloat16 (BF16) precision, a 16-bit floating-point format that maintains the same dynamic range as 32-bit floats while reducing memory usage. This makes it highly suitable for mixed-precision training, enabling faster computations and lower resource consumption without compromising model performance. BF16 achieves this by slightly reducing precision compared to FP32, but retaining the same exponent range, which ensures numerical stability and significantly enhances training efficiency-particularly for large-scale models.

To further optimize memory usage during training, a memory-efficient optimizer-paged\_adamw\_8bit-was utilized, allowing the efficient handling of large-scale tokenized datasets. All training sessions were conducted within a single Colab runtime, with automatic checkpointing to Google Drive for seamless progress tracking and recovery.

These configurations demonstrate that it is feasible to fine-tune a domain-specific language model effectively using publicly available tools and cloud-based platforms. The use of Google Colab Pro+ with A100 GPU provided the necessary performance without the need for expensive dedicated hardwares.

## Fine-Tuning with QLoRA

To efficiently fine-tune the StarCoderBase-1B model on Colab Pro+ A100 GPU, this project used QLoRA (Quantized Low-Rank Adaptation), a parameter-efficient fine-tuning (PEFT) technique. QLoRA combines 4-bit quantization with low-rank adapters (LoRA), allowing training large models with limited computational resources.

### Quantization Configuration

We used the BitsAndBytes library to quantize the base model to 4-bit precision (NF4), allowing the model to be loaded into memory efficiently and trained with minimal hardware constraints.

### LoRA Adapter Configuration

To enable adaptation on a minimal number of trainable parameters, **LoRA adapters** were applied to key linear projection layers of the transformer architecture:

from peft import LoraConfig

peft\_config = LoraConfig(

    task\_type="CAUSAL\_LM",

    inference\_mode=False,

    r=6,

    lora\_alpha=16,

    lora\_dropout=0.05,

    target\_modules=["c\_attn", "c\_proj", "c\_fc", "c\_mlp"]

)

This configuration ensures that only ~9% of the model's parameters are trainable, resulting in approximately **0.81 GB of GPU memory usage** during training.

### Training Arguments and Strategy

The model was fine-tuned using the Hugging Face Trainer API, which provided an efficient and robust framework for training large language models. All training arguments were configured to optimize both convergence and stability, given the dataset size and available GPU resources.

The core configuration is shown below:

from transformers import TrainingArguments

training\_args = TrainingArguments(

    output\_dir="/content/drive/MyDrive/final\_finetuned\_model",

    per\_device\_train\_batch\_size=6,

    gradient\_accumulation\_steps=2,

    num\_train\_epochs=2,

    learning\_rate=1e-4,

    lr\_scheduler\_type="cosine",

    warmup\_steps=100,

    fp16=True,

    eval\_strategy="steps",

    eval\_steps=1000,

    save\_strategy="steps",

    save\_steps=1000,

    save\_total\_limit=5,

    logging\_steps=50,

    logging\_dir="./logs",

    report\_to="tensorboard",

    run\_name="starcoder\_finetune\_v1",

    push\_to\_hub=False,

    load\_best\_model\_at\_end=True,

    metric\_for\_best\_model="eval\_loss",

    greater\_is\_better=False,

    gradient\_checkpointing=True,

    gradient\_checkpointing\_kwargs={"use\_reentrant": False}

)

This setup allowed for stable training on Google Colab Pro+ with a single A100 GPU, enabling efficient resource usage while maintaining robust convergence and reproducibility throughout the fine-tuning process.

### Callbacks and Monitoring

During training, custom callback functions were integrated to enhance monitoring and ensure robust checkpointing:

* **PrintLossCallback:** At the end of each epoch, this callback prints both training and validation loss values directly to the console. This provided clear, real-time insight into model convergence and overfitting without relying on external visualization tools.
* **Checkpointing:** Model checkpoints were saved automatically at the end of each epoch and at regular intervals during training. Up to five checkpoints were retained to conserve disk space while preserving recovery options in case of runtime interruptions or Colab session timeouts.

This configuration enabled stable, memory-efficient training and ensured model progress was reliably tracked and recoverable throughout the fine-tuning process.

## Evaluation Metrics and Results

A two-stage evaluation strategy was adopted to evaluate the effectiveness of the fine-tuned model in generating embedded C code. First, loss-based metrics were collected during training to evaluate the convergence of the model. Then, 10 different embedded development prompt scenarios were used to compare the functional code outputs between the pre-trained StarCoderBase-1B model and the fine-tuned version.

### Training and Validation Loss

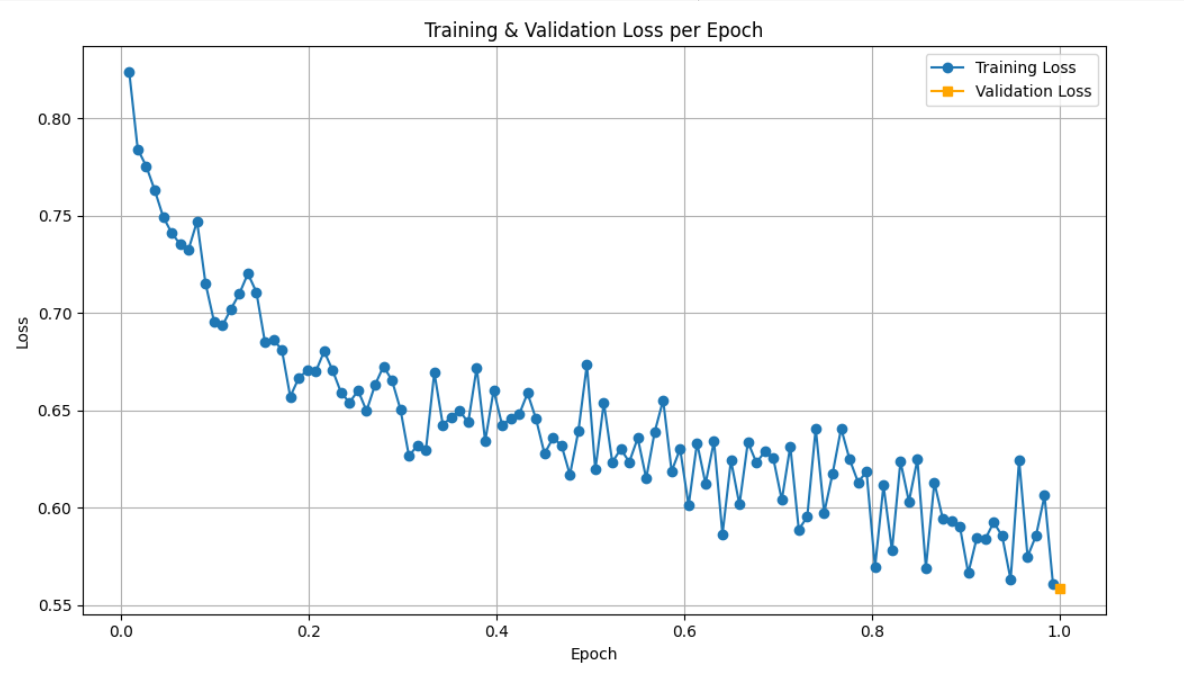


Figure 1

Figure 3.5.1, shows the training and validation loss during the training process. The training loss decreased continuously throughout the training and the model showed stable learning without any signs of overfitting, reaching a training loss of 0.6001 and a validation loss of 0.5441.

### Prompt-Based Evaluation of Embedded Peripheral Scenarios

To evaluate the fine-tuned model's effectiveness in generating embedded C code, a curated set of task-specific prompts was constructed. These prompts were primarily designed to reflect real-world scenarios commonly encountered in embedded systems. Tasks included:

* LED toggling using HAL\_Delay()
* UART-based message transmission
* Arduino-based analog read and Serial.print() every 500 ms
* PWM signal generation with TIM1
* I2C-based temperature sensor reading
* DMA-based USART data transfer
* Timer-based LED toggle using TIM2 callback

While the baseline model produced incomplete outputs for most prompts, the fine-tuned model produced consistent code snippets. Specifically, for the Arduino prompt, the fine-tuned model produced a valid framework, demonstrating adaptability across embedded platforms.

Generated outputs were assessed manually by comparing results from both the base and fine-tuned models according to the following criteria:

**Syntactic Validity:** Whether the generated code was free of syntax errors.

**HAL API Usage:** Proper inclusion and usage of STM32-specific HAL functions.

**Behavioral Coherence:** Logical alignment of the generated code with the intended functionality of the prompt.

Due to the platform-specific nature of STM32 code and the absence of vendor-specific header files (e.g., stm32f1xx\_hal.h) in general-purpose compilers such as GCC, automatic compilation was not feasible. Therefore, code correctness and platform compatibility were evaluated through manual inspection and domain-specific knowledge rather than direct compilation.

This prompt-based evaluation confirmed that the fine-tuned model significantly outperforms the base model in both task-specific accuracy and embedded hardware logic coherence.

### Comparative Analysis of Base and Fine-Tuned Model Outputs

To illustrate the practical benefits of domain-specific fine-tuning, a comparative analysis was conducted between the base and fine-tuned model outputs for a selected prompt. The task involved generating an Arduino-compatible program to toggle an LED every 500 milliseconds using digitalWrite() and delay() functions.

Prompt:

"Generate a complete Arduino sketch with setup() and loop() functions to blink the built-in LED on pin 13 every 500 milliseconds using digitalWrite() and delay()."

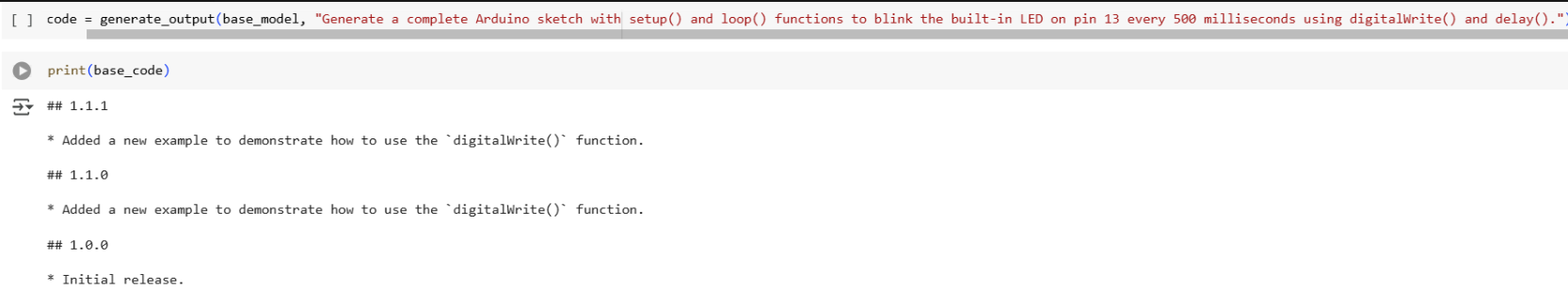


Figure 2

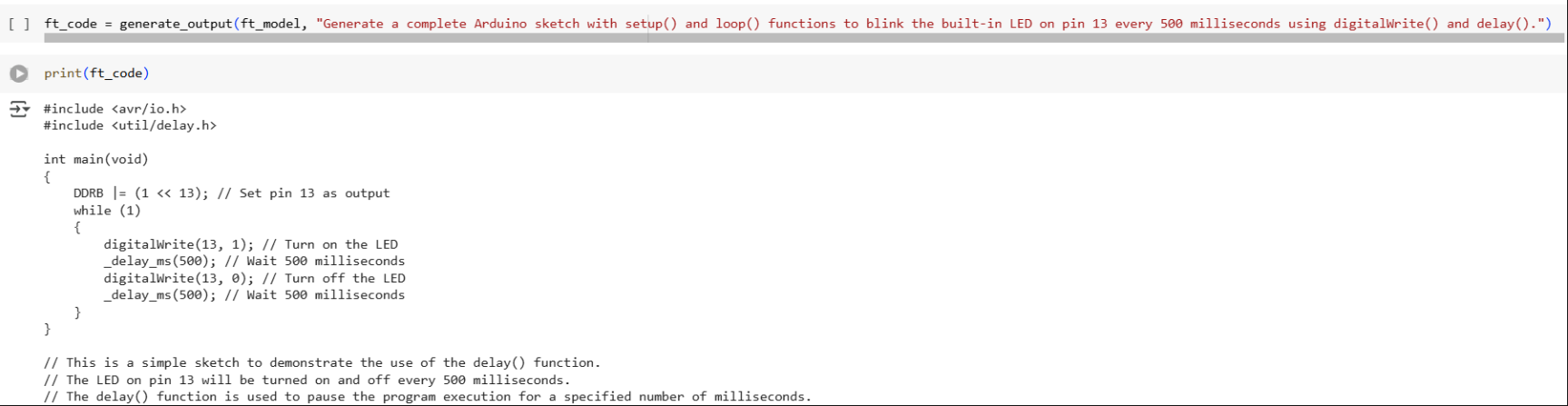


Figure 3

As shown in Figure 3.5.2, the base model failed to produce meaningful embedded code. Instead, it returned generic changelog content unrelated to the programming task, demonstrating a lack of domain alignment. This suggests that, without fine-tuning, the model lacks the contextual grounding necessary for task-specific embedded development.

However, the fine-tuned model (see Figure 3.5.3) produced the appropriate header files (<avr/io.h>, <util/delay.h>) and valid AVR C code to control the LED.

These comparisons and results demonstrate that a successful fine-tuning process has been achieved. While the pre-trained model showed limited awareness of embedded system structures, the fine-tuned version exhibited syntactic accuracy and domain-specific proficiency.

## Challenges Encountered During Implementation

A two-week literature review was conducted at the beginning of this project. Although there were similar studies, no prior work focusing solely on fine-tuning a model for embedded system C code generation was found. The uniqueness of the project introduced several challenges during its development phase.

Initially, embedded C code samples were attempted to be sourced from large-scale open-source datasets such as The Stack (BigCode, 2023), which contain code in various programming languages. However, it was observed that training performance degraded significantly due to the presence of noisy data in these datasets.

Revisions were also necessary regarding model selection. At first, the CodeLlama-7B (Rozière et al., 2023) model (with 7 billion parameters) was selected for fine-tuning. However, limitations such as insufficient dataset size and high computational cost led to a reconsideration. As a result, the focus shifted toward smaller-scale models, and StarCoderBase-1B was ultimately chosen.

To overcome this issue, a new dataset was constructed directly from open-source GitHub repositories using a custom Python pipeline. This approach involved systematically searching for and downloading only .c files whose filenames contained embedded-system-related keywords, such as "stm32", "gpio", and "hal". Additional filters, such as minimum code length, C-style comment checks, hash-based duplicate removal, and license text cleaning, were applied to maximize dataset quality. As a result, a domain-specific collection of 19,769 unique embedded C code samples (69,785 lines) was compiled, ensuring high relevance and diversity tailored for embedded system development

During training, efforts were made to fully utilize the A100 GPU on Google Colab Pro+. However, several runtime crashes due to "Out Of Memory" errors were encountered, triggered by either underutilization or overloading of GPU memory. Optimizer techniques and hyperparameters were iteratively tuned for optimal performance. After these trial-and-error processes, the project reached its final form and was completed successfully.

# CONCLUSION

This thesis presented an AI assistant designed to generate syntactically correct and hardware-oriented C code for embedded system developers. Embedded systems are widely used in areas such as automotive electronics, industrial automation, healthcare devices, and IoT-based monitoring systems. The reliability and efficiency of the software in these fields are critical.

The project addressed core challenges in embedded software development, including the complexity of hardware interactions, repetitive coding processes, and privacy concerns that arise with cloud-based AI tools. To solve these problems, a local and privacy-focused AI system was implemented. The StarCoderBase-1B model was chosen for its balance of performance and computational efficiency.

A key contribution of this study was the creation of a dataset specific to embedded C code. The dataset was constructed directly from open-source GitHub repositories using a custom Python pipeline. The pipeline used targeted keyword searches and filename filtering to identify .c files relevant to embedded systems. Quality control steps included a minimum line count, C-style comment checks, duplicate detection through MD5 hashing, and removal of legal or license texts. As a result, 19,769 unique code samples with a total of 69,785 lines were collected and prepared for model training.

Fine-tuning was performed using the QLoRA method with 4-bit quantization on a single A100 GPU in Google Colab Pro+. Both the training and validation loss values demonstrated smooth and stable convergence, indicating effective learning.

The fine-tuned model was evaluated with prompt-based tests focused on typical embedded programming scenarios, including GPIO, UART, ADC, and DMA. The model produced syntactically correct and context-appropriate C code, outperforming the base model in embedded logic consistency. These results show that the model is a practical tool for embedded developers.

Although the project proved the effectiveness of local LLM-based AI assistants for embedded system software, further improvement is possible. Future work can focus on supporting more embedded platforms and programming languages, improving security validation, and enabling interactive debugging.

In summary, this thesis demonstrated that a specialized and locally deployed language model can support embedded software development, increase productivity, and enhance code quality. The findings point to a promising direction for integrating AI-powered tools into embedded software engineering.

## Future Works

While this study demonstrated the feasibility and effectiveness of locally fine-tuned LLMs for embedded C code generation, several directions remain for future research and development:

* **Security-Aware Code Generation**: Integrating static analysis tools or training the model with security-labeled datasets can enhance the generation of safe and secure embedded code. This is particularly crucial for systems operating in critical domains such as automotive or medical devices.
* **Scaling to Larger Models and Datasets**: Training larger versions of StarCoder or similar models using more diverse and extensive datasets could improve generalization, robustness, and support for a wider range of hardware abstraction layers and coding styles.
* **Cross-Platform Embedded Support**: Future work could explore expanding beyond STM32 and Arduino platforms to include other popular microcontroller families (e.g., ESP32, MSP430, PIC), allowing the assistant to serve a broader developer base.
* **Interactive Debugging and Code Explanation**: Enabling features that allow the model to provide inline comments, explain its code reasoning, or guide developers through debugging processes can significantly improve usability and educational value.
* **User Interface and Developer Tooling**: A practical graphical user interface (GUI) or lightweight plugin can be developed to interact with the model more intuitively. Such a tool could provide prompt suggestions, code visualization, inline explanations, and one-click export to IDEs. This would make the AI assistant more accessible and usable for embedded developers with varying levels of experience.

# REFERENCES

**BigCode.** (2023a). StarCoderBase-1B [Pretrained model]

Retrieved from; <https://huggingface.co/bigcode/starcoderbase-1b>

**BigCode.** (2023b). The Stack v1.2 [Dataset].

Retrieved from; <https://huggingface.co/datasets/bigcode/the-stack>

**Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., … & Zaremba, W.**

(2021). *Evaluating large language models trained on code.* Retrieved from; <https://doi.org/10.48550/arXiv.2107.03374>

**Englhardt, Z., Miller, A., Khalid, H., … et al.** (2023). *Exploring and characterizing large*

*language models for embedded system development and debugging.* Retrieved from;

<https://doi.org/10.48550/arXiv.2307.03817>

**Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, L., … & Chen, W.** (2021).

*LoRA: Low-rank adaptation of large language models.*

Retrieved from; <https://arxiv.org/abs/2106.09685>

**Li, J., Zhang, X., Wang, Q., … et al.** (2024). An exploratory study on fine-tuning large

language models for secure code generation.

Retrieved from; <https://doi.org/10.48550/arXiv.2408.09078>

**Li, R., Tunstall, L., BigCode Team, … et al.** (2023). StarCoder: May the source be with you!

Retrieved from; <https://doi.org/10.48550/arXiv.2305.06161>

**Pearce, H., Ahmad, W., He, S., … et al.** (2022). Asleep at the keyboard? Assessing the

security of GitHub Copilot’s code contributions. In IEEE Symposium on Security and Privacy (S&P). Retrieved from; <https://doi.org/10.48550/arXiv.2108.09293>

**Rozière, B., Izacard, G., Piktus, A., Hesslow, D., Rabe, M. N., Goyal, N., … & Lample, G.**

(2023). Code Llama: Open foundation models for code. Retrieved from; <https://arxiv.org/abs/2308.12950>

# APPENDIX A

**Github link for project:** https://github.com/denizmeteee/Graduation\_Project

# CV:

**Name & Surname:** Deniz Mete Coşkunsu

**E-mail:** deniz.m.coskunsu@gmail.com

**Phone:** +90 536 601 31 40

**LinkedIn:** www.linkedin.com/in/denizmetec