SOFTWARE THAT MAKES CORRECTIONSUGGESTIONS TO ERROR DETECTED CODE BLOCKSOR FUNCTIONS*

*Note: Sub-titles are not captured in Xplore and should not be used

1st Mehmet Sıddık Aktaş

2st Ali Buğday
Computer Engineering
Yildiz Technical University
19011076
Istanbul, Turkey
ali.bugday@std.yildiz.edu.tr

3nd Çağatay Alptekin
Computer Engineering)
Yildiz Technical University
20011622
Istanbul, Turkey
cagatay.alptekin@std.yildiz.edu.tr

***Abstract*—Software and test engineers put a great deal of importance on the identification and examination of error-detected code particles along the coding process. Large language models (LLMs)-based improved models will be providing new code suggestions for code blocks that have been identified as having bugs. Software engineers will find it simpler to interact with and produce more precise and efficient codes as a result. These developments in LLM-based models accelerate the procedure, improving the quality of the code and optimizing the software development workflow as a whole.**

Our project consists of a structure that makes code suggestions using LLM models, allows comparing previously produced codes according to test results, and also allows users to test the code they produce in two different languages. In this way, software, test engineers and all kinds of users will be able to compare these models and generate code by choosing the model among the options.

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

The introductory chapter describes the basic problem that our project aims to solve. The chapter consists of two subchapters. The first chapter focuses on the difficulty that exists in the everyday life of a target group. The second chapter deals with a short collection of ideas and the subsequent introduction to the rough idea of the core project.

A. Fundamental Problem

In their regular job, software developers often come across flaws in their code. It would be a big help to their workflow and productivity if they had a tool that suggested code to fix these mistakes using various models. They will be able to get code suggestions from various models and they can compare the results and test it as they want. The aim of this project is to enhance software development processes by generating better code suggestions and enabling code comparison and testing through the use of Large Language Models (LLMs). By employing LLM-based models to more effectively identify and analyze error-detected code blocks, software engineers can generate code that is both more correct and efficient. Along with encouraging collaboration, the project gives users the tools they need to write well-informed code by allowing them to test code in multiple languages and compare code suggestions from different models.

Such a tool would accelerate debugging while promoting cooperation and information sharing within development teams by giving developers smart, context-aware suggestions for repairing code bugs. Such a tool will be a big help when programmers encounters code errors and doesn't know how to solve them. They can gaze into the test results that were given to the them for comparison goals to select the best option for their needs and they can test this code results generated by the program, to see how accurate this generated code. Our project aims to see the results of different LLM's. Users of this program will have an idea of how different LLM's respond to different questions. With the help of our predefined questions and tests, user can see before running any code, how different the outputs generated from different LLM's. Project also visualizes the results from different models with graphs for users to easily differentiate which model seems better. In the end, it would enable programmers to produce high-caliber software more quickly and with cleaner, more reliable code.

B. Motivation

By helping coders in generating code, LLM's can improve general productivity. LLM's can serve as a learning point for developers, meaning that developers can inspect the generated code from the LLM's and compare with their code generation ability and they can gain new insights about code generation. Also, even if the code is error free, LLM's still could be useful in a way that showing alternative ways to achieve similar results. Our project aims not only generate codes using different models, but also the testing ability to see and

Identify applicable funding agency here. If none, delete this.

compare the results of the codes. We have chosen **Python** and **Javascript** to coding languages to generate code blocks or functions.

1) The Way The Program Works: Initially user picks the model they want to run the command with the coding language selection. Program displays a text field for the prompt. user generates a code with a prompt, then copies this code and adds it to the testing procedure with his own parameters. Testing results are displayed to the user. In this way users can test the models with very different parameters evaluate different models. Previously, ten distinct questions are used to test each model, five of which are easy and the other five of which are more difficult.

Every question contains 10 different test parameters and results. With the help of previously given tests, the user will be able to see how successfully the model has passed the existing tests, without needing to provide additional test parameters. Additionally, the user will have the ability to compare using the graphs we provide. This will allow them to have a better idea of the model's performance. For the test, the **Unit Test** methodology is applied.

C. Project Scope

The **codellama**, **llama**, and **mistral** are the LLM's that are used amd tested locally for performance comparison purposes. Local functionality has been made possible via the development of reduced versions.

Furthermore, **Colab Pro** has been used to run and create code for huge models. **deepseek-coder-6.7b**, **CodeLlama-13B**, and **WizardCoder-33B** are some of these models. There is also **Mistral-8x7B**. These models are obtained from **huggingface**.

II. Related Work

A. LLM-based Test-driven Interactive Code Generation

In this study, a unique interactive approach called TICODER is presented, which guides intent clarification through tests to improve the accuracy of code generation. It demonstrates significant improvements in code accuracy and reduced cognitive load through user studies and testing with cutting-edge large language models. [9]

B. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation

In order to overcome shortcomings in current benchmarks, the study presents EvalPlus, a methodology for thoroughly assessing the functional correctness of code synthesized by Large Language Models (LLMs). This is achieved by enhancing evaluation datasets with automatically produced test cases. [10]

C. A Performance Study of LLM-Generated Code on Leetcode

In comparison to human-crafted solutions on Leetcode, the paper assesses how effective Large Language Models (LLMs) are at generating code. It finds that LLM-generated code performs comparably across models and is typically more efficient than human-written code, with implications for future optimizations in code generation. [11]

D. Studying LLM Performance on Closed- and Open-source Data

This paper investigates the performance gap of Large Language Models (LLMs) in proprietary software development versus open-source projects. It finds that, although LLMs have little effect on performance in C# code, their application to C++ code results in a significant reduction in performance because of identifier differences. [12]

E. Evaluating Code Generation by Learning Code Execution

The paper presents CodeScore, a unique Code Evaluation Metric (CEM) that takes functional equivalency into account and supports diverse input formats, thereby addressing shortcomings of current match-based metrics. [13]

F. EvaluLLM: LLM assisted evaluation of generative outputs

In this paper, the application EvaluLLM—which formulates evaluations as choices between pairs of generated outputs based on user-defined criteria—is introduced. EvaluLLM uses large language models (LLMs) to simplify and automate the evaluation of natural language generation (NLG) outputs, providing an alternative to traditional metrics and expensive human evaluation. [14]

G. Comparison of Large Language Models in Answering Immuno-Oncology Questions: A Cross-Sectional Study

This study compared the performance of ChatGPT-4, ChatGPT-3.5, and Google Bard in answering questions related to immuno-oncology, finding that ChatGPT-4 and ChatGPT-3.5 outperformed Google Bard in terms of completeness, accuracy, relevance, and readability of responses. [15]

H. A Survey on Evaluation of Large Language Models

This paper offers a thorough analysis of large-scale language model (LLM) evaluation techniques, emphasizing the significance of evaluating LLM performance on a range of tasks and the ramifications for society. It highlights how important it is to approach assessment as a basic discipline in order to support the growth of more skilled LLMs. [16]

I. Performance evaluation of ChatGPT, GPT-4, and Bard on the official board examination of the Japan Radiology Society

In responding to clinical radiology questions from the Japan Radiology Board Examination (JRBE), this study assesses the performance of large language models (LLMs), such as ChatGPT, GPT-4, and Google Bard. It finds that GPT-4 significantly beat both ChatGPT and Google Bard. [17]

J. Large language model applications for evaluation: Opportunities and ethical implications

This paper explores the emergence of Large Language Models (LLMs), like ChatGPT, emphasizing how these models might transform text-based activities in domains like assessment. [18]

K. Large Language Models as Test Case Generators: Performance Evaluation and Enhancement

The effectiveness of Large Language Models (LLMs) in producing high-quality test cases is investigated in this research, which concludes that LLMs have computational constraints when it comes to complicated situations. [19]

L. "Which LLM should I use?": Evaluating LLMs for tasks performed by Undergraduate Computer Science Students

This study assesses several large language models (LLMs), including Microsoft Copilot, GitHub Copilot Chat, Google Bard, and ChatGPT(3.5), for a range of tasks in undergraduate computer science education in India. [20]

M. Large Language Model Evaluation Via Multi AI Agents: Preliminary results

In order to assess and compare the performance of various Large Language Models (LLMs), this paper presents a novel multi-agent AI model that retrieves code based on descriptions from different LLMs and evaluates their outputs using the HumanEval benchmark. [21]

N. HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in Hugging Face

HuggingGPT, an AI agent driven by large language models like ChatGPT, is proposed in this paper as a way to orchestrate and integrate different AI models available on platforms like Hugging Face. [22]

O. A Systematic Review of Testing and Evaluation of Healthcare Applications of Large Language Models (LLMs)

This research emphasizes how little real patient data sets are used to evaluate Large Language Models (LLMs) in the healthcare domain, with the majority of studies concentrating on tasks like as question answering and medical knowledge assessment, mostly in internal medicine and surgery. [23]

P. MEGA: Multilingual Evaluation of Generative AI

The performance of generative language models is compared to non-autoregressive models in this paper's introduction of MEGA, the first thorough benchmarking of generative language models across 70 languages. Through a variety of exercises, it examines these models' strengths and weaknesses. [24]

Q. Using Large Language Models to Provide Formative Feedback in Intelligent Textbooks

This paper presents models that use principal component analysis to provide formative feedback on summaries written by students in intelligent textbooks. Compared to previous approaches, these models achieve significantly higher variance explanation for content and wording. [25]

R. Evaluating Large Language Models for Structured Science Summarization in the Open Research Knowledge Graph

Using Large Language Models (LLMs) such as GPT-3.5, Llama 2, and Mistral, the work proposes automating the process of proposing characteristics to define research contributions and compares its performance with manually selected attributes from the Open Research Knowledge Graph (ORKG). [26]

S. Large Language Models As Annotators: A Preliminary Evaluation For Annotating Low-Resource Language Content

The purpose of this work is to investigate how multilingual large language models (LLMs) can help or replace human-generated annotations in low-resource languages such as Indic languages. [27]

T. Evaluating Large Language Models: A Comprehensive Survey

The goal of this paper is to guide the development of comprehensive evaluation frameworks by discussing the multifaceted evaluation of Large Language Models and highlighting the need for thorough assessment in areas like knowledge, alignment, and safety in order to maximize their potential while minimizing risks like data leaks and misinformation. [28]

III. Technical Knowledge

This chapter serves to describe the components and software in detail, explaining all factual aspects. The chapter covers the three main topics, each with its own focus. The focus is explicitly explained in the subchapter, followed by the models used with their basics. The focus in this chapter is to review previous technical knowledge and to supplement missing information. All necessary and project-relevant details are described in detail and explained with graphics.

A. Software

1) Python:

2) Colab:

B. Completion of the technical summary

In this chapter, a detailed overview was given of which models and softwares were used in the project and which software was used for programming. In addition to the detailed description of the individual components, alternatives were also provided which offered themselves alongside the chosen tool. With each decision, the alternatives were presented, thus revealing the scope of the available options.

Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.

IV. Definitions

The world of code generation utilizes some powerful tools and technologies. Before delving into our approach, It's critical to first understand the fundamental concepts. This section will explain the prospects of Large Language Models (LLMs), the importance of platforms such as Hugging Face, and the critical technology of Transformers, upon which LLMs are created. We'll also explain how these powerful models might be applied locally for research reasons. By reading these fundamental concepts, you can have a solid foundation for understanding the process underlying LLm world.

A. Large Language Models (LLMs)

Large Language Models are powerful AI systems that have been trained on large volumes of text. They are skilled in understanding and producing natural language, translating languages, writing many types of creative text formats, and even generating different kinds of code. LLMs learn using a technique known as supervised learning, in which they are exposed to large datasets of text and code examples during training. This enables them to find patterns and relationships in the data, resulting in content that is both contextually relevant and grammatically acceptable.

LLMs have broad applications in a variety of sectors. They are used to power chatbots, improve search engines, aid in content production, and, more crucially for our research, generate various code architectures. Their ability to comprehend and respond to complicated instructions makes them an attractive tool for automating code generation work. In our study we focused on thir code generating skills. [1]

1) Brief History of Large Language Models (LLMs): In recent years, LLMs have evolved rapidly. A significant milestone was reached in 2017 with the introduction of the transformer architecture, which has since become a fundamental building block for many modern LLMs.This breakthrough paved the way for models like BERT, which has gained widespread adoption for its language understanding capabilities.

The GPT series of models, developed by OpenAI, has been particularly influential. GPT-2 garnered attention for its impressive text generation abilities, sparking discussions around responsible use of such powerful technology. Later versions like GPT-3 and ChatGPT have gained even more acclaim for their multi-modal capabilities, generating public interest and sometimes media hype.

While some companies have focused on developing large, closed-source models, the open-source community has played a critical role in democratizing access to LLMs. Projects such as BLOOM, LLaMa, and Mistral demonstrate the potential of open-source LLMs, with some even outperforming market leaders.

2) Training and Architecture: Below are some often used expressions to describe the structure and training of LLMs:

- **Supervised Learning and Fine-Tuning:**Supervised Learning and Fine-Tuning: LLMs begin by learning from large text and code datasets using supervised learning, which trains them to predict the next word or character in a sequence. They can then be fine-tuned for specific tasks by performing additional training on specialist datasets. This deliberate refining improves their performance in specific applications. Techniques such as ”reinforcement learning from human feedback” (RLHF) help to steer this process by allowing human preferences to influence the model's outputs.

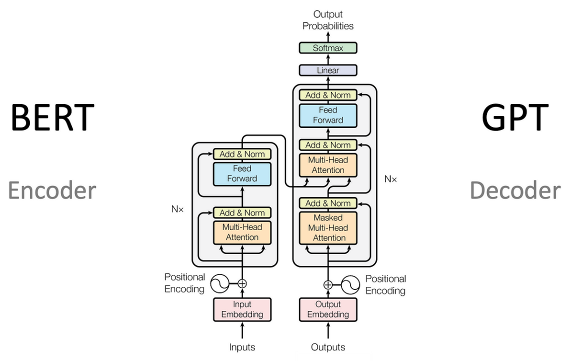


Fig. 1. All language models (LLMs) have a certain structure. Some use an encoder architecture (e.g., BERT), some use a decoder architecture (e.g., GPT), and others combine the two. [7]

- **Transformer Architecture:** The transformer design is a fundamental building piece for many current LLMs. It allows models to comprehend the relationships between words and code elements in a stream of data. Transformers, using mechanisms such as self-attention, can capture long-range dependencies and contextual information. The architecture consists of an encoder and a decoder; the encoder processes input sequences to generate contextual embeddings, which are then used by the decoder to produce outputs, making transformers extremely effective for natural language and code interpretation applications.
- **Attention Mechanism:** This mechanism within the transformer helps the LLM determine which parts of a text or code sequence are most relevant to each other, assigning ”attention weights' accordingly.
- **Context Window:**LLMs process input text and code in segments called context windows. The size of this window impacts how much information the model can consider at once when generating responses.

3) Evaluation of LLMs: Several methods are used to assess the performance and capabilities of Large Language Models. Here are some important concepts and metrics:

- **Perplexity:** A fundamental metric, perplexity measures how well the LLM predicts a given sequence of text or code. A lower perplexity score indicates a better predictive ability.
- **Task-Specific Datasets and Benchmarks:** LLMs are evaluated using specialized datasets designed to test skills like question answering, text completion, and coding. Benchmarks that combine various tasks offer a broader assessment of an LLM's capabilities.

- **Zero-Shot vs. Few-Shot Evaluation:** An LLM may be tested directly (”zero-shot”) or provided with a few examples of how to perform a task (”few-shot”) before being evaluated. Researchers use a variety of techniques to craft the prompts used in these tests.

V. Methodology

In this section, we will go over the methods we used to generate code with Large Language Models (LLMs), create test suites, and assess the performance of these models for tackling various coding difficulties. Our strategy comprises picking the most appropriate LLM, generating coding tasks, creating test cases, establishing assessment criteria, and organizing results visualization. The goal of this methodology is to evaluate the capability of LLMs at various code complexity levels. By doing so, we hoped to contribute to the literature on LLMs and their possible uses in software engineering, particularly automated code creation.

A. Running LLms on local environment

Large Language Models are extremely strong, but their large computational needs make them challenging to use. In this section, we talked about how to run these models on local system.We will examine the methods to run these(huggingface, llama etc).In addition, we’ll look at optimization approaches like quantization and pre-trained quantized models, which allow us to run even larger models in limited resources environments.

1) Tools and Frameworks: Here are some options for running LLM’s in a local environment.

- **Hugging Face Transformers:**The Hugging Face Transformers library stands as an invaluable resource in the world of Large Language Models. We used this option in our project with the instructions of our project consultant. It has several important functions:
  - *Centralized LLM Repository:*Hugging Face hosts a vast collection of pre-trained LLMs, including popular open-source models and those developed by major research institutions. This makes it a convenient starting point to find and experiment with suitable models for every project.
  - *Tools for Transformation and Adaptation:*The Transformers library goes beyond providing models. It includes tools for loading and fine-tuning, manipulating LLMs for various tasks. This grants us the flexibility to optimize LLMs for our specific needs or test their performance under different conditions.
  - *Community-Driven Innovation:* Hugging Face supports a thriving community of researchers and developers. This collaborative atmosphere promotes information sharing, the development of custom models, and rapid progress in the sector.
  - *key advantages of HF:* The library is designed to be user-friendly, making it accessible to persons with limited prior LLM knowledge.Hugging Face Transformers provides a degree of standardization in how models are shared and used, improving reproducibility and collaboration in the LLM research community. it olso includes more models than other options
- **Libraries for Local Execution:**OLLAMA, lama.cpp

2) Optimization Techniques::

- *Quantization:* Quantization is a model compression approach that decreases the memory footprint of large language models (LLMs) by transforming their weights and activations from high-precision to low-precision forms. This method entails converting 32-bit floating-point numbers (FP32) into more compact representations, such as 8-bit or 4-bit integers (INT8 or INT4). Quantization considerably reduces the model’s overall size, allowing it to run on less powerful hardware. Image compression is a good comparison for learning quantization because lowering the size of an image requires deleting some information while keeping acceptable quality. Similarly, quantizing an LLM reduces its precision while keeping the loss of accuracy within acceptable limits. This leads to models that are smaller, use less memory, require less storage, and can perform inferences faster. As a result, quantized models can run on a broader range of devices, including single GPUs and even CPUs, rather than being restricted to high-end hardware with multiple GPUs. One commonly used quantization technique is the affine quantization scheme, which maps high-precision FP32 values to lower-precision integer values using scaling factors. This method starts by calibrating the model with a smaller dataset to determine the range of FP32 values, which are then scaled down and rounded to the nearest integer. While quantization can occasionally result in a loss of accuracy, this can be reduced by quantizing weights in blocks, which reduces the influence of outliers while preserving higher precision.
- *Pre-Quantized Models:* Several libraries and platforms provide pre-quantified models customized for local circumstances, allowing users to reap the benefits of quantification without having to go through the complicated procedure. These repositories provide ready-to-use models that have been rigorously quantized to balance performance and memory efficiency, allowing for easy integration into a variety of projects.
  One significant example is the Hugging Face Model Hub, which has a large number of pre-quantified models. These models are intended to run smoothly on local PCs with modest hardware resources. The Hugging Face community is constantly updating and optimizing these models to keep them cutting-edge while also making them more accessible to a wider audience.
  Another useful resource is the GGML (Georgi Gerganov Machine Learning) library, which specializes in quantifying Llama models for CPU execution. The library supports a variety of quantization techniques, including

mixed precision approaches that maximize efficiency and accuracy. GGUF (GPT-Generated Unified Format), GGML's successor, extends similar functionality to non-Llama models by providing a flexible and extensible framework for running quantized models on a wide range of hardware configurations.
likewise, the GPTQ (General Pre-Trained Transformer Quantization) method uses a layer-wise quantization strategy to reduce output error, allowing models to run on a single GPU with lower memory needs. This method is especially beneficial for preserving model performance while dramatically lowering resource use. Users can easily build efficient, high-performance LLMs using pre-quantized models from established repositories and platforms, eliminating the need for substantial hardware investments or in-depth technical understanding of the quantification process. [8]

3) Hardware Considerations:

- *Minimum Requirements:* We have shown the hardware features in the computer environment we used in this study and the hardware features in the collap environment in the tables. We recommend at least 16 GB of RAM for running a 7b parameter model on your local PC. You will have approximately 10 GB of free space, especially if you are using an operating system like Windows that consumes the majority of your RAM. Only quantized models with 7b or smaller parameters can be run in this section. In contrast, we encountered no RAM shortages for these models in the collap environment.
- *GPU Acceleration:* Large Language Models (LLMs) benefit greatly from GPU acceleration. The massive computational demands of LLMs can be effectively controlled by exploiting the parallel processing capabilities of GPUs. GPUs are built to handle thousands of threads at once, dramatically lowering the time necessary for training and inference. This acceleration is especially crucial for models with billions of parameters, such as GPT-3.5, which need significant computational resources. The speed and efficiency of GPUs allow for faster iterations and experimentation, which is critical in research and development. likewise, GPUs provide scalability and energy efficiency, which are critical for managing the large-scale matrix operations and data processing that LLMs require. Modern GPUs now offer mixed-precision training, which improves efficiency while preserving accuracy.
- *Trade-offs:* When working with Large Language Models (LLMs), balancing model size, computational resources, and inference performance requires substantial tradeoffs. Larger models, while more capable and accurate, require a lot of memory and processing capacity, which often necessitates high-end GPUs. This raises both the cost and energy consumption. Smaller models, particularly those that use techniques such as quantization, use less memory and resources but may have worse accuracy and capabilities. Using greater processing resources can enhance inference speed, but it also increases operational expenses. Finding the right mix is critical for delivering successful and efficient LLM-based apps.

Display card	GTX 1650
Display card VRAM	4GB
RAM	16GB
CPU	intel i7

TABLE I
THE COMPUTER ON WHICH WE RUN THE MODELS IN THE LOCAL ENVIRONMENT

Type	VRAM	RAM
Cpu	none	12.7GB, 50GB(If high ram option is selected)
T4	15GB	
V100	16GB	
L4	22GB	

TABLE II
THE HARDWARE WE USE IN THE COLLAB ENVIRONMENT

B. How did we select LLms?

Beginning our project without prior knowledge of Large Language Models presented a challenge in choosing suitable models for code generation.Our research led us to explore popular models accessible through the Hugging Face platform. Time constraints limited our ability to do extensive testing on different models.We eventually selected three instruct-tuned 7B models quantized as GGuf to meet our hardware limitations:

- **Llama-7b:**Developed by Meta Platforms, Inc., LLama offers solid code generation capabilities. We focused on the 7B variant, given our hardware constraints, but it's important to note that larger versions (13B, 70B) exist. LLama models are specifically fine-tuned for chat-like interactions, with a default context length of 4096 tokens.
- **Codellama-7b:**Built upon LLama 2 and fine-tuned for code-related interactions, Codellama was selected to explore its potential for improved code generation compared to the base LLama model. It has the capacity to both generate code and provide natural language explanations across a variety of popular programming languages.
- **Mistral-7b:**Developed by the French AI company Mistral AI. Mistral-7B is a 7.3B parameter language model utilizing the Transformer architecture. It showcases innovative design choices like Grouped-Query Attention (GQA), contributing to its impressive performance. Benchmarks demonstrate that Mistral-7B often outperforms larger LLMs like LLama (13B, 34B variants). Additionally, Mistral AI offers larger models under the Mixtral 8x7B designation. [2]

We initially looked at the Falcon model.However, the need for more easily available quantized versions compatible with our hardware setup resulted in its elimination.
Our project's exploratory approach aligned with our focus on open-source, 7B quantized models. Because of their improved

capacity to engage with the user and obey directions, the instruct-tuned character of these models was enticing.

C. Implementation of Models

During the first week, we spent our time learning the subject by using educational platforms such as YouTube, Udemy, EdX, and Medium. We also uploaded a model named ”llama2-7b” to the collab and received answers to the prompts we gave.
In the second week, we were able to download and execute models like ”llama,” ”mistral,” and ”falcon” in ggml format on our local envirement using hugging face.
The third week was dedicated to testing the models. We researched on how ”llms” are tested and identified and tested the tests in the section that we explained in the testing phase.
In the fourth week, we ran larger models and different quantization types like ”gptq” and ”awq” in a collab environment. Unfortunately, we were unable to test and compare these models due to time constraints.
During the fifth week, we began developing an application that allows users to interact with the models via the interface. This process took two weeks since we built more detailed tests.

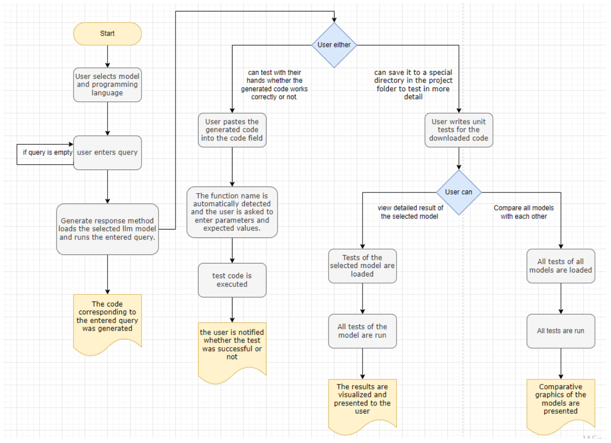
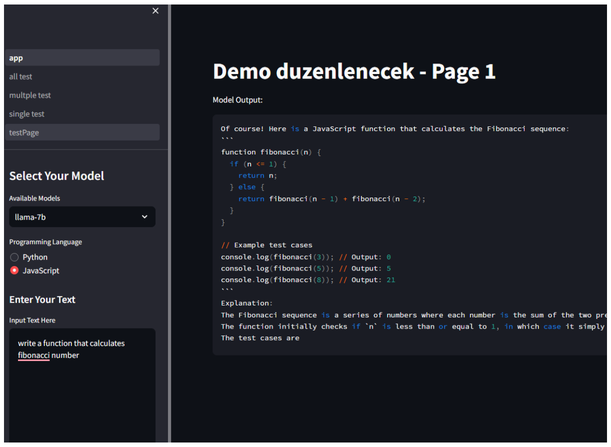

Start
User selects model and programming language
if query is empty
user enters query
Generate response method loads the selected llm model and runs the entered query.
The code corresponding to the entered query was generated
User either
can test with their hands whether the generated code works correctly or not.
can save it to a special directory in the project folder to test in more detail
User pastes the generated code into the code field
The function name is automatically detected and the user is asked to enter parameters and expected values.
test code is executed
the user is notified whether the test was successful or not
User writes unit tests for the downloaded code
User can
view detailed result of the selected model
Compare all models with each other
Tests of the selected model are loaded
All tests of the model are run
The results are visualized and presented to the user
All tests of all models are loaded
All tests are run
Comparative graphics of the models are presented

Fig. 2. flowchart showing how the application works(yazilar gozukmuyor tekrardan duzenlenecek)

Our app makes it easier to engage with LLMs for code generation by providing a user-friendly four-page interface. It enables users to select their preferred LLM, choose between Python or JavaScript, and generate code specific to their query. The platform provides flexibility in reviewing results, allowing users to manually test the generated code within the interface or download it for complete unit testing in their own development environment. This multi-faceted approach caters to diverse user needs, providing both convenience and in-depth testing options for LLM-generated code.
The core functionality resides in the getresponse_model_name function, responsible for interacting with various LLMs. This function initially loads the user-specified model (e.g., Codellama-7b) from a specific location, along with parameters such as model type and GPU use.Next, it constructs a prompt tailored to the user's query and chosen programming language by leveraging a customizable template. Finally, the function uses the LLM's capabilities by feeding it the formed prompt, resulting in code that matches the user's input. This modular design, combined with user interface integration for input validation and progress indication, improves the code generation process for user.



Demo duzenlenecek - Page 1

Select Your Model

Available Models: llama-7b

Programming Language: Python [✓ JavaScript]

Enter Your Text

Input Text Here: write a function that calculates fibonacci number

Model Output:

```
Of course! Here is a JavaScript function that calculates the Fibonacci sequence:
```
function fibonacci(n) {
  if (n <= 1) {
    return n;
  } else {
    return fibonacci(n - 1) + fibonacci(n - 2);
  }
}

// Example test cases
console.log(fibonacci(3)); // Output: 0
console.log(fibonacci(5)); // Output: 5
console.log(fibonacci(8)); // Output: 21
```
Explanation:
The Fibonacci sequence is a series of numbers where each number is the sum of the two pre
The function initially checks if `n` is less than or equal to 1, in which case it simply
The test cases are
```

Fig. 3. interface of the application's homepage and sample code generation

One of the problems with our application is that it responds late, which is perfectly normal because the models run on the CPU. This situation caused us difficulties in testing. Waiting 2 minutes for an answer to each query made it difficult to ask the models a total of 90 queries. To solve this problem, the first thing to do was to try the GPQ qunatize models running on the GPU VRAM. We were able to run these models locally after many errors, which we ran smoothly in the collap environment, but we could not see a significant change in terms of speed. Another problem we had with the GPQ models was that we did not have enough GPU VRam to run these models. Our computer's GPU has 4 GB VRAM, which means that our 7B models would not be enough even though the size was compressed. This version led us to our second option: we moved the same models from our local model to the collap environment, and instead of loading the model and running the query each time, loading the model first and then running the queries one by one created a serious time advantage.

D. design tests

As we mentioned in the similar project section, testing llms is a major challenge. First, we'd want to present the findings from our literature review on the subject, and then we'll describe how we created the tests.
LLM benchmarks provide an established structure for assessing the capabilities of Language Models (LLMs) in various language-related tasks, ranging from question answering to code creation. These benchmarks include precisely created tasks, questions, and datasets that aim to objectively compare different LLMs and track the growth of individual models over time. The evaluation method involves providing tasks to the model, assessing its performance using measures such as accuracy, BLEU score, or perplexity IV-A3, and

Type	Number of questions we asked the models at each level	Number of tests written for one answer	Total assertion
easy	10	5	10X5=50
medium	10	5	10X5=50
difficult	10	5	10X5=50
			150

including human evaluation for more nuanced assessments. LLMs encounter benchmarks through different paradigms, including zero-shot, few-shot, and fine-tuned approaches, each revealing unique insights into the model's adaptability and learning capacityIV-A3. Notably, benchmarks for coding problems examine LLMs' capacity to generate functional code. Benchmarks like HumanEval [3], MBPP [4], and SWE-bench [5] assess skill in problem-solving and real-world software engineering tasks.

Our project consultant requested us to evaluate the models at three different levels: easy, medium, and difficult. The challenge with testing the models was determining the questions from sites where the problems were categorized by level. We collected our tests from the Leetcode website. Example questions we asked based on the levels: In the basic level, we asked models to find the greatest common divisor of two given numbers (gcd); in the medium level, we asked models to sort a given array with merge sort; and in the difficult level, we tried to test it with queries like coin change. That is 10 questions at each level, and we had to analyze 30 questions for each model, for a total of 90. In fact, since the models provided different answers to a question each time, we allowed each query the ability to execute three questions. In the tests we created, we accepted whichever of these three responses produced the best results. In truth, we could fix this problem by setting the model's temperature parameter to 0, ensuring that it gives the same answer to every query but when we set the temperature to zero, we could not get good results in these models, according to our observations. It's important to highlight our testing process involved experimenting with various parameter values such as temperature, top_k, and repetition_penalty to identify optimal settings for our models. Different models exhibit varying performance with specific parameter configurations, underscoring the need for customization. Notably, the usage advice part of the Hugging Face models' documentation specifies recommended parameter values. However, for fairness and consistency, we ultimately opted to evaluate all three models using their default parameters(We did not set parameters). As we wrote in the previous section, time was a big problem because at first it took us 2 minutes to get answers to each query. We solved this problem by first loading the model in the collap environment and then running the queries one by one.

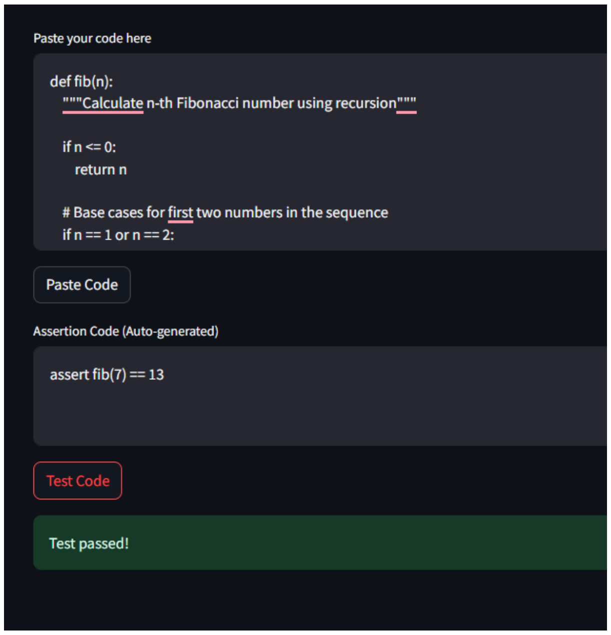
The part so far has been about how we produce answers to queries. From now on, it will be about how we test answers to queries. Frankly speaking, these two parts were where most of our time went.In our testing methodology, after generating replies for each query using our models, we used unit tests to validate the accuracy of these results carefully. Taking inspiration from software testing principles, we constructed a suite of unit tests tailored to assess the accuracy and reliability of the answers. For example, consider query 1, which is an easy query with the objective of computing Fibonacci numbers. We created unit tests to verify the model's reactions for various input values, including 0, 1, 7, -5, and 52. Each test case uses the assertEqual method to compare the model's function output to the expected result.By running these unit tests, we carefully examined the model's performance across a wide range of inputs, ensuring that it consistently generates accurate and dependable results. This careful testing method not only confirms the model's responses, but also instills confidence in its ability to handle a wide range of scenarios and query types.

E. visualizing results

In the visualization phase of our study, we used Python's Plotly package to create captivating visual representations of our results. Using Plotly's features, we built dynamic and interactive visualizations such as pie charts, bar charts, and stacked bar charts. These visualizations served as effective tools for conveying complex information in a clear and intuitive manner.Additionally, to provide users with a comprehensive overview of any faulty functions identified during our analysis, we presented this information in a structured table format. This allowed users to rapidly identify and resolve any concerns raised by our study.
In terms of the graphical user interface (GUI) development, we explored several alternatives before settling on Streamlit.Despite the initial learning curve connected with Streamlit, we discovered that its simplicity and ease of use greatly accelerated the GUI development process. Streamlit's straightforward UI and wide widget library sped up the construction of our user interface, allowing us to concentrate on our application's functionality and features. While mastering Streamlit presented some challenges, its efficiency and versatility ultimately proved invaluable in bringing our project to fruition.

VI. Results

In the results section, we provide a full analysis of the outcomes of our thorough evaluation and comparison of the models under consideration. Before diving into the specifics of our findings, it's worth emphasizing one critical feature of our testing technique. We created a user-friendly test page to allow for hands-on evaluation of the code generated by the models. This page serves as a practical tool for users to validate the executability of the code before proceeding with detailed testing. The procedure is simple: users copy the code generated on the main page, paste it onto the test page, and the program will automatically detect the function name. Subsequently, users are prompted to input parameters and expected values, and the system promptly informs them of the test's success or failure. While we acknowledge that this system may have certain security vulnerabilities, we assure users that it operates within a local environment and is designed solely for users benefit. This proactive approach empowers users to ensure the functionality and reliability of the generated code, contributing to a more robust and informed testing process.


Paste your code here

```
def fib(n):
  """Calculate n-th Fibonacci number using recursion"""

  if n <= 0:
    return n

  # Base cases for first two numbers in the sequence
  if n == 1 or n == 2:
```

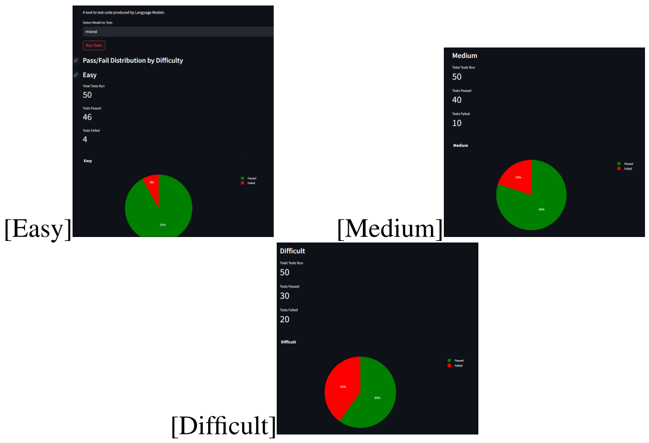
Assertion Code (Auto-generated)

```
assert fib(7) == 13
```

Test passed!

Fig. 4. Test page where the user can quickly test the generated code with by hands.

our tool automatically collects and visualizes the aggregate results of unit tests, providing users with a comprehensive perspective of model performance. Users can determine the frequency of successful and unsuccessful unit test executions using simple visuals. If users want to dig deeper into model performance, they can select a specific model and run all associated tests. This allows users to assess the model's success rate at all tested levels, providing vital insights into its overall efficacy, as seen in figure 5. In addition, users can obtain the model's cumulative success rate across all tiers


A tool to test code produced by Language Models

Select Model to Test:

mistral

Easy
Total Tests Run
50
Tests Passed
46
Tests Failed
4

Medium
Total Tests Run
50
Tests Passed
40
Tests Failed
10

Difficult
Total Tests Run
50
Tests Passed
30
Tests Failed
20

Fig. 5. The detailed performance of a selected model is initially presented using pie charts.

using the brief presentation of a stack bar chart in Figure 6. To further enhance user understanding, a detailed table highlighting functions where the model exhibits incorrect behavior is furnished (figure 7). This holistic approach provides users with the knowledge they need to make informed decisions about model selection and utilization, resulting in a more efficient and effective testing process.

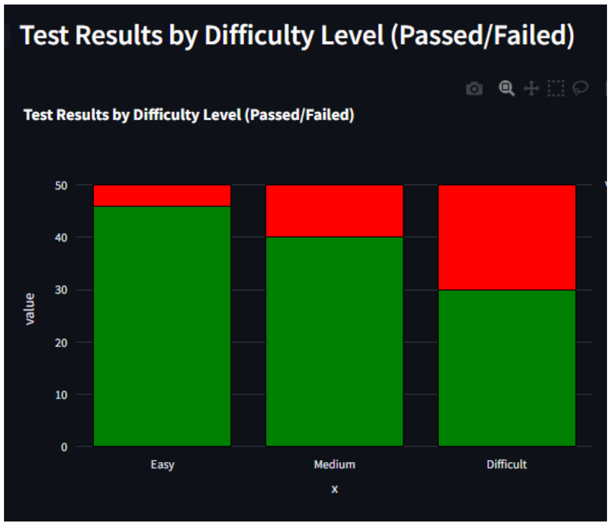
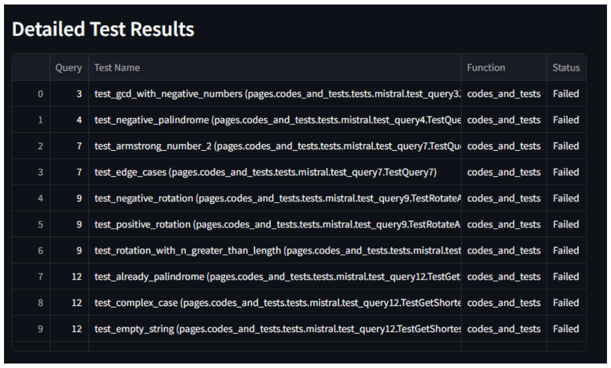


Fig. 6. Summarizing the performance of the models displayed on the pie chart in terms of the levels on the stacked bar chart


Detailed Test Results

	Query	Test Name	Function	Status
0	3	test_gcd_with_negative_numbers (pages.codes_and_tests.tests.mistral.test_query3.	codes_and_tests	Failed
1	4	test_negative_palindrome (pages.codes_and_tests.tests.mistral.test_query4.TestQue	codes_and_tests	Failed
2	7	test_armstrong_number_2 (pages.codes_and_tests.tests.mistral.test_query7.TestQu	codes_and_tests	Failed
3	7	test_edge_cases (pages.codes_and_tests.tests.mistral.test_query7.TestQuery7)	codes_and_tests	Failed
4	9	test_negative_rotation (pages.codes_and_tests.tests.mistral.test_query9.TestRotateA	codes_and_tests	Failed
5	9	test_positive_rotation (pages.codes_and_tests.tests.mistral.test_query9.TestRotateA	codes_and_tests	Failed
6	9	test_rotation_with_n_greater_than_length (pages.codes_and_tests.tests.mistral.test	codes_and_tests	Failed
7	12	test_already_palindrome (pages.codes_and_tests.tests.mistral.test_query12.TestGet	codes_and_tests	Failed
8	12	test_complex_case (pages.codes_and_tests.tests.mistral.test_query12.TestGetShorte	codes_and_tests	Failed
9	12	test_empty_string (pages.codes_and_tests.tests.mistral.test_query12.TestGetShortes	codes_and_tests	Failed

Fig. 7. The names of malfunctioning functions are presented to the user

After thoroughly examining each model's performance in isolation, we proceeded to compare them with each other. Initially, we presented the success rates of the models across different difficulty levels, detailing their performance in easy, medium, and difficult tasks as (fig: 8).Following that, we conducted a thorough comparison by contrasting the models' overall success rates across all difficulty levels (as shown in Figure 9). This approach enabled a thorough evaluation of the models' performance across a wide range of task difficulties, providing clear insights into their respective strengths and weaknesses.

The results we received validate the comparison findings presented on this webpage [6]. Based on these results, the

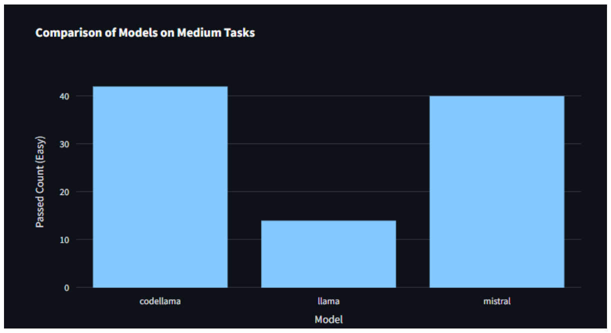


Fig. 8. Graph showing the comparison of models with each other in intermediate level questions

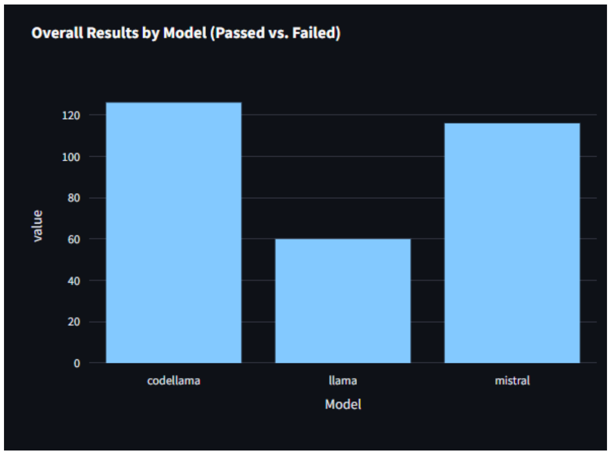
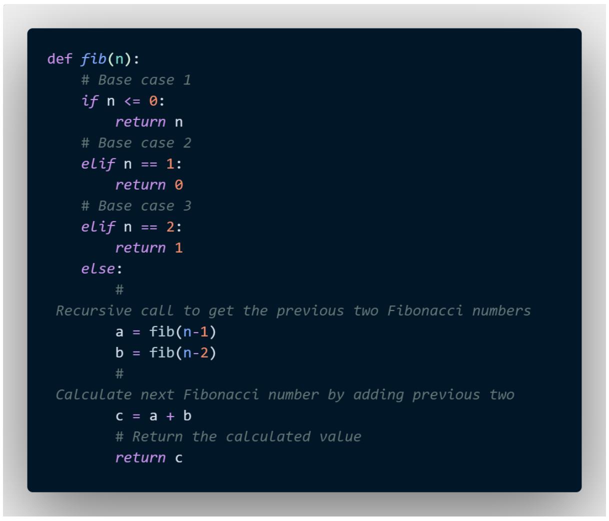


Fig. 9. Graph showing models comparing aggregate results across all levels

Codellama model demonstrates superior performance, outperforming the llama model. This is an expected result because the Codellama model is fine-tuned on the Llama model and is a specialized for coding. Meanwhile, the Mistral model achieves performance levels comparable to Codellama, just like as shown on the our results.

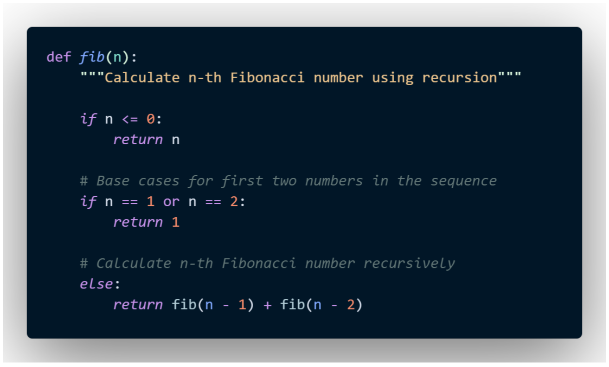
A. Code Quality Analysis

The fact that LLms produced different answers each time made us think at first. This situation is due to parameters such as temperature in the measurements. The more this parameter is given a value other than zero, this means that you can take this much risk while responding to the model. We asked the Mistral model to calculate the Nth fibonacci number, and while the first code it generated (showed in fig10) failed, the second code it generated (11) worked successfully. We were initially unsure how we might test in such a situation. Setting the temperature to zero seemed a sensible answer, instructing the model to provide the expected result without taking any risks, but this time we could not get good performance from our models. As we said in the methodology section, we handled this problem by allowing each model to run it three times and accepting whichever one produced the best results. In this way, we believe we can measure their abilities more fairly without exposing them to the risks they take.



```
def fib(n):
    # Base case 1
    if n <= 0:
        return n
    # Base case 2
    elif n == 1:
        return 0
    # Base case 3
    elif n == 2:
        return 1
    else:
        #
 Recursive call to get the previous two Fibonacci numbers
        a = fib(n-1)
        b = fib(n-2)
        #
 Calculate next Fibonacci number by adding previous two
        c = a + b
        # Return the calculated value
        return c
```

Fig. 10. The incorrect answer given by the mistral model to the fibonacci question



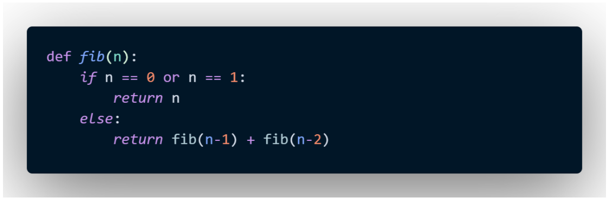
```
def fib(n):
    """Calculate n-th Fibonacci number using recursion"""

    if n <= 0:
        return n

    # Base cases for first two numbers in the sequence
    if n == 1 or n == 2:
        return 1

    # Calculate n-th Fibonacci number recursively
    else:
        return fib(n - 1) + fib(n - 2)
```

Fig. 11. The correct answer given by the mistral model to the fibonacci question



```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Fig. 12. The correct answer given by the codellama model to the fibonacci question but not as good as mistral

Unit tests serve as a valuable method for assessing LLM models' performance. As the number of tests increases, models tend to approximate real-world results more closely, yet they may not show the whole picture. For instance, while the answer depicted in picture 12 may appear to be the optimal solution for the Fibonacci question within the coding model, it exhibits a flaw—entering an infinite loop when provided with a negative number. In contrast, the Mistral model, while behind slightly in test performance, tackles this issue by integrating the response illustrated in Figure 11 and using comment lines to improve code readability. Despite our efforts to push each question to its limits in the unit tests, it is imperative to underscore the importance of human evaluations in unit testing. Ensuring code understandability alongside functionality remains paramount, highlighting the necessity for human intervention in the testing process. Ensuring code understandability alongside correctness of the code remains paramount, highlighting the necessity for human intervention in the testing process.

VII. Discussion and Conclusion

Throughout the project, the codes generated by the models consistently demonstrated high quality. Moreover, even the easy questions we asked the models (gcd, palindrome number) were actually questions that even some university students could not do. We had to push the models in order to test their performance as thoroughly as feasible. Medium and difficult level questions were ones that could be answered in a few stages. Notably, while the LLAMA model performed similarly to other models on simple problems, it struggled when faced with medium and harder challenges. Overall, Codellama outperformed Mistral marginally. However, while choosing between these models, we prefer Mistral. While Codellama is optimized for code creation, Mistral's versatility extends beyond that, surprising us with its proficiency in coding domain while being a general-purpose model.

When running LLms, especially in the local environment, the limited hardware environment limits the models you can use. Here the quantization method came to our aid. Quantization emerges as a valuable solution to address these limitations, allowing for the reduction of model size while incurring only a minor decrease in performance. For example, while a model with 7 billion parameters would normally require 14 gigabytes of RAM, quantization allows the model to operate within a substantially smaller memory footprint. Although we did not use quantized models built for full GPU use, such as GPTQ, it is worth emphasizing their potential benefits, especially for applications that require high-speed interactions with users in server-based systems. Notably, the increasing performance of smaller models, exemplified by models like Mistral, has become noteworthy, as they are capable of rivaling or surpassing larger counterparts while remaining viable for deployment on low-level devices. As a result, in recent years, we can see that firms building LLm are attempting to lower the size of their models while enhancing performance. In practice, running the models used in our project in a local environment requires at least 16 GB of RAM. However, for maximum productivity and cost-effectiveness, using cloud-based environments like Google Colab emerges as a best practice. Running the models here takes less time than training, therefore it will not strain your budget.

In the field of application interface development, we created an intuitive interface that allows for seamless user interaction. Using the Streamlit library in the graphical user interface (GUI) component, we discovered this technology to be extremely effective and recommend it to others embarking on similar initiatives. Streamlit provides the advantage of designing powerful interfaces with less code, hence speeding the development process and increasing productivity. While the technology may be difficult to understand in some areas, particularly the session component, the overall benefits transcend these minor complexity. Furthermore, by rigorously comparing and validating LLM models against existing benchmarks, we belive that we have made an important contribution to the literature. By correlating model comparisons discovered online with our empirical findings and publishing our findings through this publication, we hope to contribute to the collective knowledge base and inform future research and development efforts in this area.

Although our study spanned a duration of six weeks, it provided a robust foundation for our future endeavors in this technology. Initially, incremental improvements can be achieved through parameter adjustments and prompt engineering interventions. However, for significant enhancements, expanding the scope of testing is essential. Automation of the testing process becomes critical to reducing the time-consuming nature of manual testing. Categorizing tests by topic, such as string manipulation or dynamic programming, can provide insights into model performance across multiple issue domains. We took our questions from the leetcode website. These questions are good when it is come to measuring algorithmic ability, but it is also important to test these models according to real-world problems instead, perhaps one of the models was trained only for the such problems in leetcode, as real-world problems require different solutions. We'd like to learn how to fine tune the models as part of this project, but we'll need an appropriate dataset and suitable equipment first. We underlined the relevance of human evaluation in the preceding results section, and we can explore for ways to incorporate human evaluation with unit tests in future investigations. For example, using technologies like langcahin, we could link previous findings to models and the outcomes they will produce, allowing us to provide feedback on the model's output. Overall, we had a lot of fun with this project and learned a lot about the technology. We are convinced that this technology will revolutionize the world in the future years by introducing innovation and automation in a variety of industries.

References

[1] Large language model. Wikipedia. from https://en.wikipedia.org/wiki/Large_language_model .
[2] Mistral AI. (n.d.). Wikipedia. https://en.wikipedia.org/wiki/Mistral_AI
[3] Papers with Code - HumanEval Benchmark (Code Generation). https://paperswithcode.com/sota/code-generation-on-humaneval
[4] Papers with Code - MBPP Benchmark (Code Generation). https://paperswithcode.com/sota/code-generation-on-mbpp
[5] SWE-bench. https://www.swebench.com/

[6] Mistral 7B LLM – Nextra. https://www.promptingguide.ai/models/mistral-7b.
[7] Norouzi, A. (2023, August 30). Discovering LLM Structures: Decoder-only, Encoder-only, or Decoder-Encoder. Medium. https://medium.com/artificial-corner/discovering-llm-structures-decoder-only-encoder-only-or-decoder-encoder-5036b0e9e88 .
[8] Talamadupula, K. (2024, February 21). A Guide to Quantization in LLMs. Symbl.ai. https://symbl.ai/developers/blog/a-guide-to-quantization-in-llms/
[9] arXiv:2404.10100 [cs.SE]
.
[10] Liu, J., Xia, C. S., Wang, Y., & Zhang, L. (2023, December 15). Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. https://proceedings.neurips.cc/paper_files/paper/2023/hash/43e9d647ccd3e4b7b5baab53f0368686-Abstract-Conference.html
[11] Tristan Coignion, Clément Quinton, Romain Rouvoy. A Performance Study of LLM-Generated Code on Leetcode. EASE24 - 28th International Conference on Evaluation and Assessment in Software Engineering, Jun 2024, Salerno, Italy. ⟨hal-04525620⟩
[12] arXiv:2402.15100 [cs.SE]
[13] arXiv:2301.09043 [cs.SE]
[14] Michael Desmond, Zahra Ashktorab, Qian Pan, Casey Dugan, and James M. Johnson. 2024. EvaluLLM: LLM assisted evaluation of generative outputs. In Companion Proceedings of the 29th International Conference on Intelligent User Interfaces (IUI '24 Companion). Association for Computing Machinery, New York, NY, USA, 30–32. https://doi.org/10.1145/3640544.3645216
[15] Giovanni Maria Iannantuono, Dara Bracken-Clarke, Fatima Karzai, Hyoyoung Choo-Wosoba, James L Gulley, Charalampos S Floudas, Comparison of Large Language Models in Answering Immuno-Oncology Questions: A Cross-Sectional Study, The Oncologist, Volume 29, Issue 5, May 2024, Pages 407–414, https://doi.org/10.1093/oncolo/oyae009
[16] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, Wei Ye, Yue Zhang, Yi Chang, Philip S. Yu, Qiang Yang, and Xing Xie. 2024. A Survey on Evaluation of Large Language Models. ACM Trans. Intell. Syst. Technol. 15, 3, Article 39 (June 2024), 45 pages. https://doi.org/10.1145/3641289
[17] Toyama, Y., Harigai, A., Abe, M. et al. Performance evaluation of ChatGPT, GPT-4, and Bard on the official board examination of the Japan Radiology Society. Jpn J Radiol 42, 201–207 (2024). https://doi.org/10.1007/s11604-023-01491-2
[18] Head, C. B., Jasper, P., McConnachie, M., Raftree, L., & Higdon, G. (2023). Large language model applications for evaluation: Opportunities and ethical implications. New Directions for Evaluation, 2023, 33–46. https://doi.org/10.1002/ev.20556
[19] arXiv:2404.13340 [cs.SE]
[20] arXiv:2402.01687 [cs.CY]
[21] arXiv:2404.01023 [cs.SE]
[22] Shen, Y., Song, K., Tan, X., Li, D., Lu, W., & Zhuang, Y. (2023, December 15). HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in Hugging Face. https://proceedings.neurips.cc/paper_files/paper/2023/hash/77c33e6 a367922d003ff102ffb92b658-Abstract-Conference.html
[23] Bedi, S., Liu, Y., Orr-Ewing, L., Dash, D., Koyejo, O., Callahan, A., Fries, J. A., Wornow, M., Swaminathan, A., Lehmann, L., Hong, H. J., Kashyap, M., Chaurasia, A. R., Shah, N., Singh, K., Tazbaz, T., Milstein, A., Pfeffer, M. A., & Shah, N. (2024, April 16). A Systematic Review of Testing and Evaluation of Healthcare Applications of Large Language Models (LLMs). medRxiv (Cold Spring Harbor Laboratory). https://doi.org/10.1101/2024.04.15.24305869
[24] arXiv:2303.12528 [cs.CL]
[25] Morris, W., Crossley, S., Holmes, L., Ou, C., McNamara, D., Dascalu, M. (2023). Using Large Language Models to Provide Formative Feedback in Intelligent Textbooks. In: Wang, N., Rebolledo-Mendez, G., Dimitrova, V., Matsuda, N., Santos, O.C. (eds) Artificial Intelligence in Education. Posters and Late Breaking Results, Workshops and Tutorials, Industry and Innovation Tracks, Practitioners, Doctoral Consortium and Blue Sky. AIED 2023. Communications in Computer and Information Science, vol 1831. Springer, Cham. https://doi.org/10.1007/978-3-031-36336-8_75
[26] arXiv:2405.02105 [cs.AI]
[27] Bhat, S., & Varma, V. (2023, January 1). Large Language Models As Annotators: A Preliminary Evaluation For Annotating Low-Resource Language Content. https://doi.org/10.18653/v1/2023.eval4nlp-1.8
[28] arXiv:2310.19736 [cs.CL]