ACE ENGINEERING COLLEGE (Department of Data Science)

AI Based Code Refactoring

Major Project Stage-2

Guided By:

Ms.Swathi Turai
(Assistant Professor)

Presented By:

P.Praneetha (22AG5A6702)
B.Aishwarya (21AG1A6710)
V.Mani Charan (22AG5A6705)
Mohammed Adil (21AG1A6744)

Contents

Introduction

- Code refactoring is the process of restructuring existing code to improve its internal structure while preserving its external behavior.
- It helps make code cleaner, more readable, and easier to maintain without adding new features or fixing bugs directly.
- Regular refactoring reduces technical debt, making the codebase more adaptable to future changes and minimizing the risk of bugs.
- It enhances collaboration by ensuring consistent coding standards and simplifying onboarding for new developers.
- Refactoring is widely used during agile development, before adding new features, or when optimizing performance and readability.

Problem Identified

- Modern software systems often suffer from code quality issues such as redundancy, poor structure, and lack of adherence to best practices. Manual code refactoring is time-consuming, error-prone, and requires deep domain expertise.

Refactoring in most current development environments is manual or semi-automated, relying on developer expertise and IDE-supported tools with limited scope and rule-based transformations.

Existing tools like IntelliJ IDEA, Eclipse, and Visual Studio offer basic refactoring features such as renaming, method extraction, and code formatting, but lack deep semantic understanding or learning-based adaptability.

These conventional systems often fail to detect complex code smells, architectural issues, or opportunities for improvement beyond syntax-level changes, leading to inconsistent and suboptimal code quality in large projects.

Existing System

VS

Proposed System

Introduces an AI-powered refactoring engine that leverages machine learning and code analysis techniques to understand code semantics and suggest or apply meaningful transformations.

Goes beyond rule-based refactoring by identifying deeper structural issues, code smells, and optimization opportunities using learned patterns from large codebases.

Aims to provide intelligent, context-aware refactoring recommendations that improve code quality, enhance maintainability, and reduce developer effort across diverse programming languages.

Proposed Solution

- This project focuses on building an AI-driven system that can automatically refactor source code to enhance readability, maintainability, and overall code quality while preserving its original functionality.

- The system leverages machine learning and natural language processing techniques to understand code semantics, identify design flaws, and perform intelligent refactoring based on learned patterns.

- It supports multiple programming languages and can detect code smells, rename variables meaningfully, simplify complex constructs, and integrate with modern development tools.

Requirement Analysis

Software Requirements

- **Operating System:** Windows 10 / Linux (Ubuntu 20.04+) / macOS
- **Programming Language:** Python 3.8+
- **IDE/Editor:** VS Code / PyCharm / IntelliJ IDEA
- **Libraries & Frameworks:**
  - TensorFlow / PyTorch (for ML models)
  - Scikit-learn (for preprocessing/classification tasks)
  - Flask / FastAPI (for building backend APIs)
- **Version Control:** Git

Hardware Requirements

- **Processor:** Intel i5 or higher / AMD Ryzen 5 or above
- **RAM:** Minimum 8 GB
- **GPU:** For training models (NVIDIA recommended).

Requirement Analysis

Functional Requirements

- The system accepts source code as input in supported programming languages such as Python or Java.
- It identifies refactoring opportunities like renaming variables, simplifying methods, and eliminating redundant code.
- Refactoring operations are performed while preserving the original behavior of the program.
- A preview of the refactored code is provided before applying any changes.

Non Functional Requirements

- The system delivers accurate and consistent refactoring outcomes with minimal false detections.
- Refactoring operations are completed within an acceptable time frame, even for moderately sized files.
- Scalability is maintained to handle large projects and multiple concurrent users.
- Compatibility is ensured across major platforms including Windows, macOS, and Linux.

System Analysis

- **Data Collection and Preprocessing:** Large open-source code repositories (e.g., GitHub, CodeSearchNet) are used as data sources. Source files are parsed to extract Abstract Syntax Trees (ASTs) and code tokens. The data is cleaned, normalized, and annotated to highlight code smells and potential refactoring opportunities.

- **Model Training and Selection:** Machine learning models (e.g., Transformers, Graph Neural Networks) are trained on the labeled dataset to learn code patterns and refactoring rules. The models are designed to predict potential refactorings or suggest changes based on historical examples.

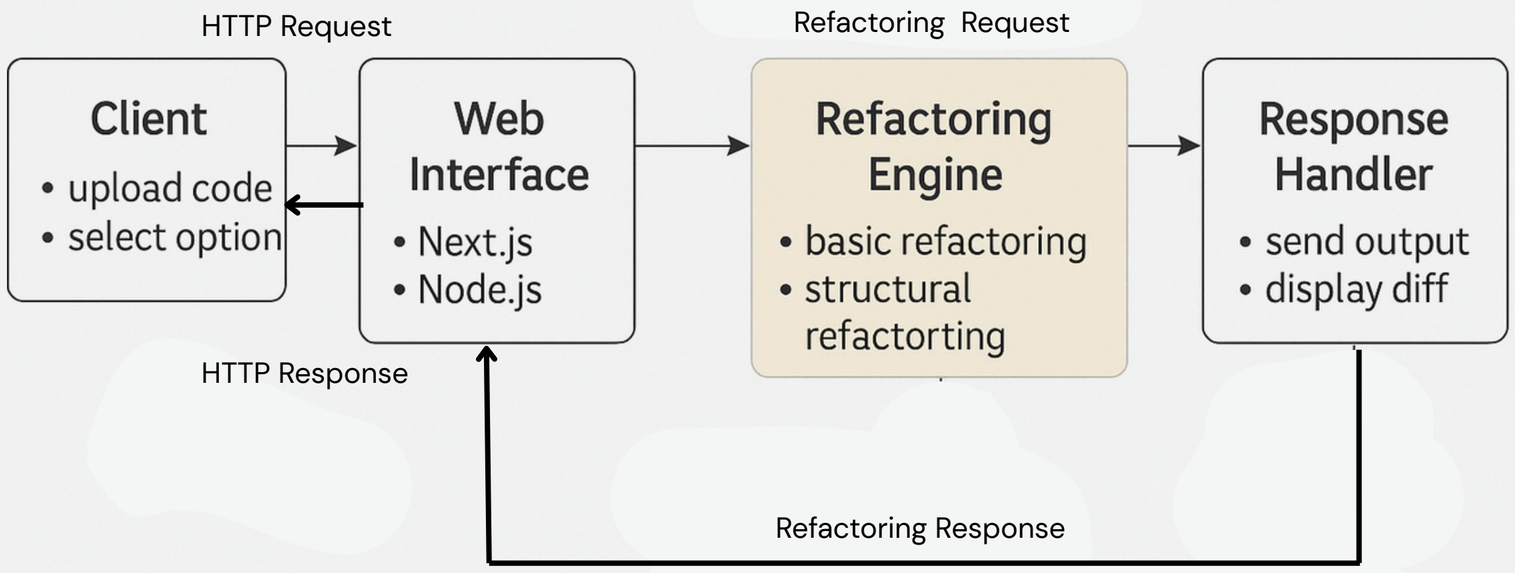
- **Refactoring Engine:** The trained model is integrated with a rule-based engine to apply or suggest refactorings. The engine ensures behavior preservation by validating that input and output code snippets produce the same results using automated test cases or semantic checks.

System Analysis

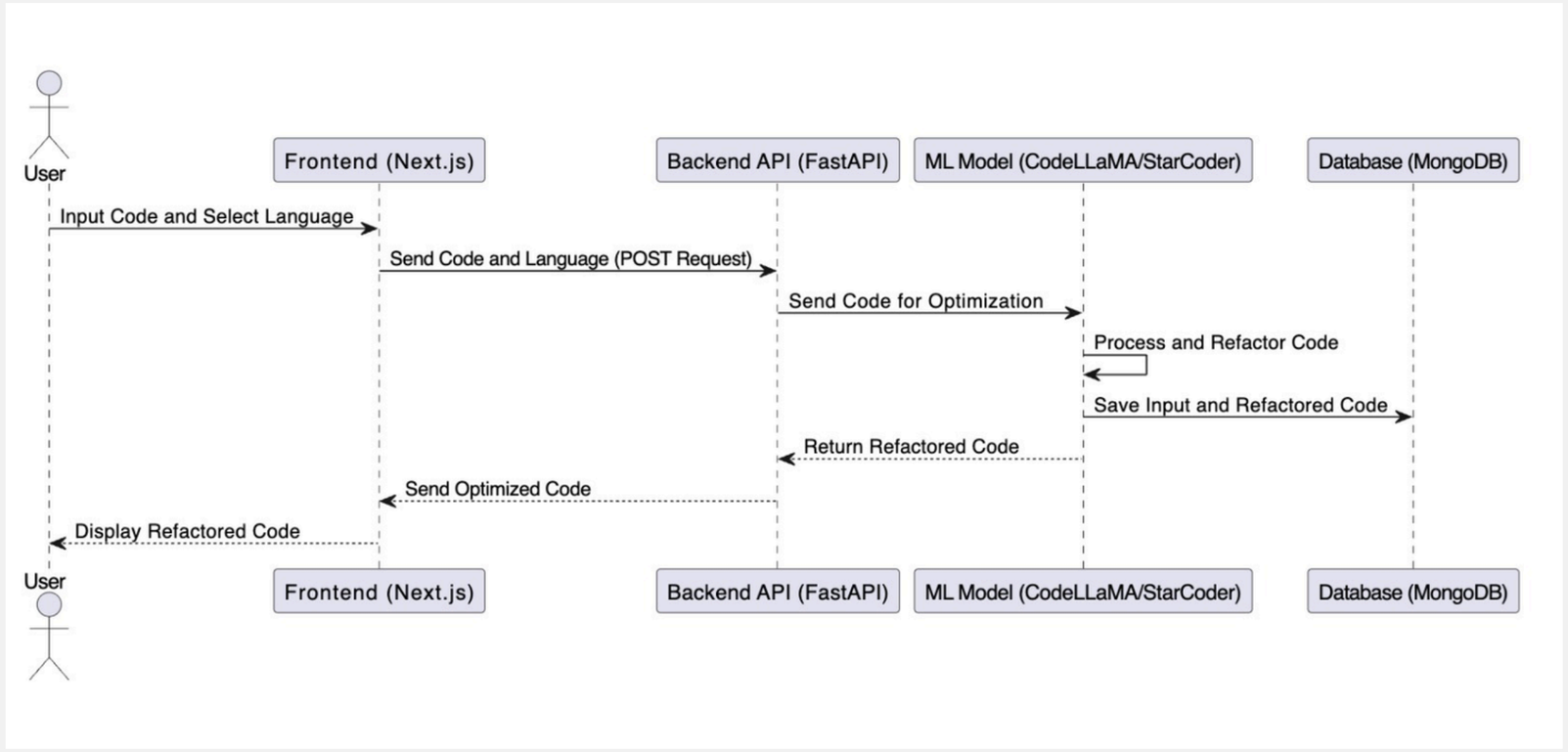
- **User Interface / API Integration:** A web-based interface or plugin is provided for users to input code and view suggested refactorings. Options are included for previewing changes, accepting or rejecting suggestions, and exporting the refactored code. Alternatively, a REST API allows integration with external development tools.

- **Evaluation and Testing:** The system is evaluated using benchmark datasets and real-world projects. Metrics such as code quality improvement (e.g., Cyclomatic Complexity, Maintainability Index), accuracy of refactoring, and user satisfaction are used to assess system performance.

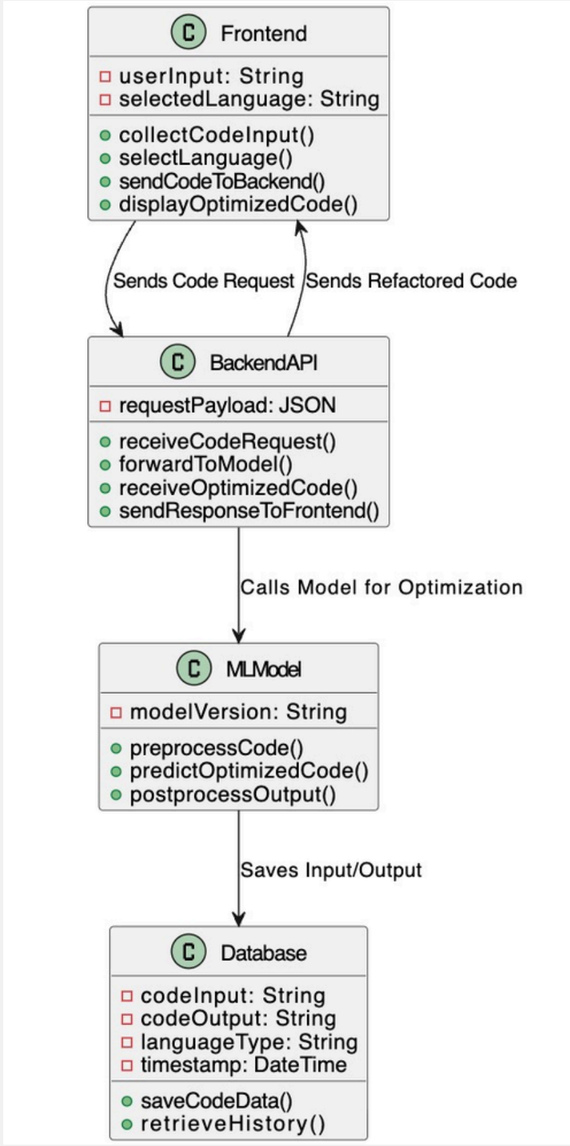
System Architecture



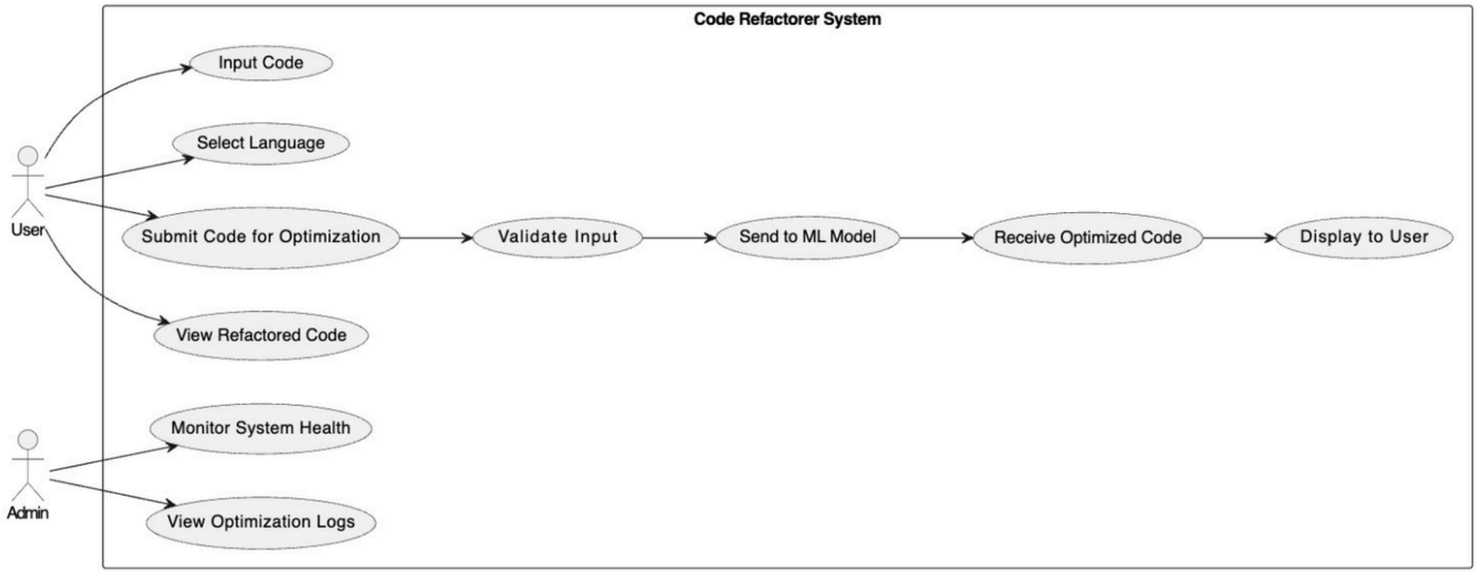
Sequence Diagram



Class Diagram



Usecase Diagram



Feature Engineering

- **Code Tokenization:** Split the input code into structured tokens such as keywords, variables, and operators.
- **Language Tagging**: Add a language-specific prefix (e.g., <lang:Python>) to guide the model's understanding of code syntax.
- **Structure Preservation:** Maintain original code indentation and formatting to retain code block meaning during preprocessing.
- **Noise Removal:** Remove unnecessary comments, unused imports, redundant spaces, and dead code to simplify inputs.
- **Special Start/End Tokens:** Add [START_CODE] and [END_CODE] markers to define clear input boundaries for the model.
- **Input-Output Pairing:** Prepare supervised datasets with (bad code → optimized code) pairs to train the model effectively.

Models Used

1. Fine-tuned CodeLLaMA

A pretrained large language model specialized for code tasks.

Fine-tuned on (bad code → optimized code) pairs to learn refactoring patterns.

2. Fine-tuned StarCoder

Another powerful model trained on large codebases.

Used for comparison or alternative fine-tuning for multi-language support.

3.CodeGen 6B:

Used as a baseline model to understand differences in optimization capabilities.

Training Set: 80% of the data used to train the model.

Testing Set: 20% of the data used to evaluate model performance.

Sample Code

```python
# backend/api.py

from fastapi import FastAPI, Request
from model import optimize_code  # assume this connects to your model

app = FastAPI()

@app.post("/optimize")
async def optimize(request: Request):
    payload = await request.json()
    code = payload.get("code")
    language = payload.get("language")

    optimized_code = optimize_code(code, language)
    return {"optimized_code": optimized_code}
```

Sample Code

```python
# backend/model.py

from transformers import AutoTokenizer, AutoModelForSeq2SeqLM

tokenizer = AutoTokenizer.from_pretrained("your-finetuned-model")
model = AutoModelForSeq2SeqLM.from_pretrained("your-finetuned-model")

def optimize_code(code: str, language: str) -> str:
    input_text = f"<lang:{language}>\n{code}"
    inputs = tokenizer(input_text, return_tensors="pt")
    outputs = model.generate(**inputs, max_new_tokens=500)
    optimized_code = tokenizer.decode(outputs[0], skip_special_tokens=True)
    return optimized_code
```

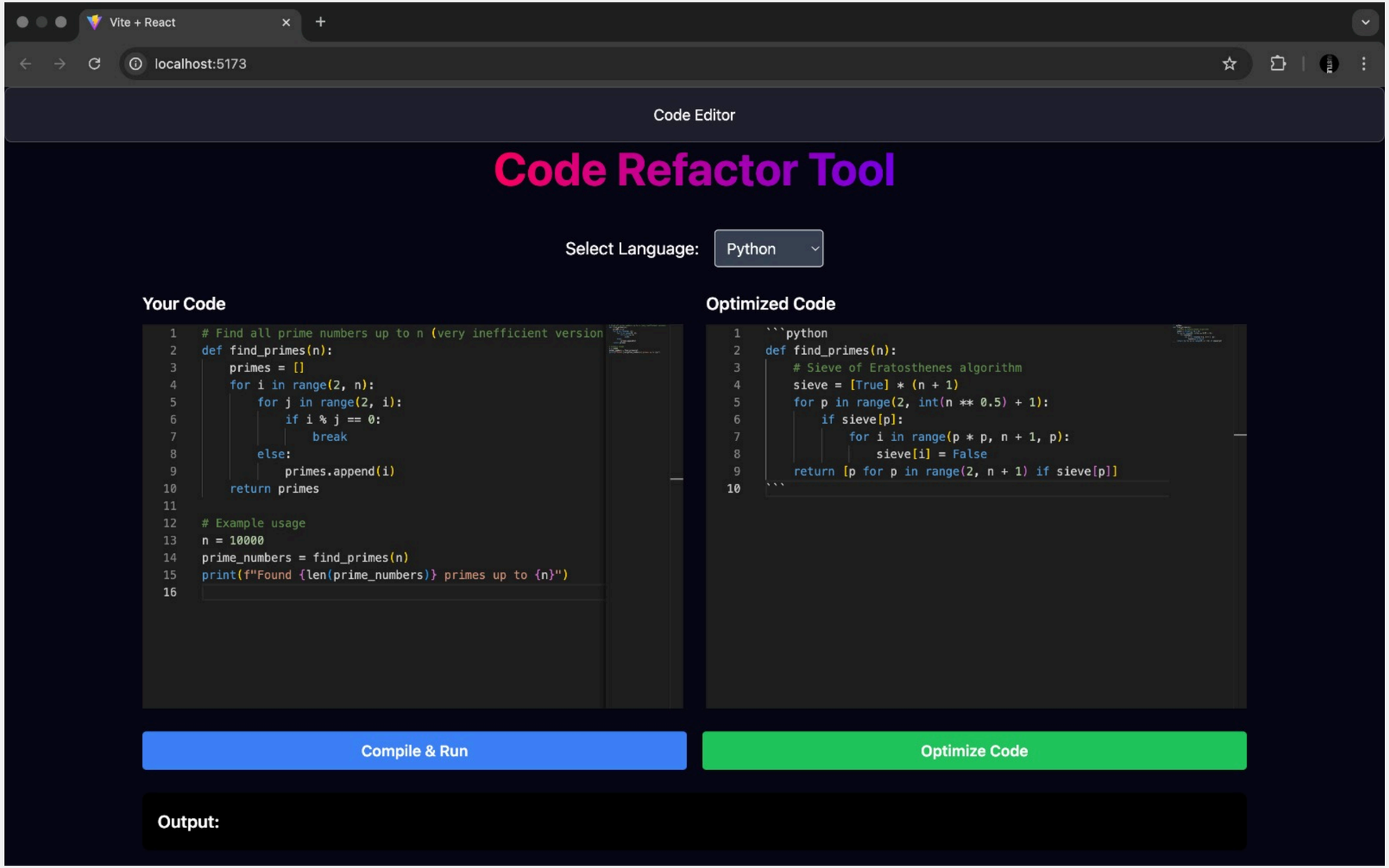
Sample Code

```
// frontend/services/api.js

import axios from 'axios';

export async function optimizeCode(code, language) {
  const response = await axios.post('http://localhost:8000/optimize', {
    code,
    language
  });
  return response.data.optimized_code;
}
```

Score Matrix

Model	Accuracy	Precision	Recall	F1-Score
Fine-tuned CodeLLaMA 7B	91.3%	90.5%	89.8%	90.1%
Fine-tuned StarCoder 7B	90.1%	88.7%	88.0%	88.4%
CodeGen 6B	88.9%	87.2%	86.5%	86.8%
Base CodeLLaMA (without tuning)	81.4%	79.9%	78.7%	79.3%

Results


Vite + React
localhost:5173
Code Editor
Code Refactor Tool
Select Language:
Python
Your Code
# Find all prime numbers up to n (very inefficient version
def find_primes(n):
primes = []
for i in range(2, n):
for j in range(2, i):
if i % j == 0:
break
else:
primes.append(i)
return primes
# Example usage
n = 10000
prime_numbers = find_primes(n)
print(f"Found {len(prime_numbers)} primes up to {n}")
Optimized Code
```python
def find_primes(n):
# Sieve of Eratosthenes algorithm
sieve = [True] * (n + 1)
for p in range(2, int(n ** 0.5) + 1):
if sieve[p]:
for i in range(p * p, n + 1, p):
sieve[i] = False
return [p for p in range(2, n + 1) if sieve[p]]
```
Compile & Run
Optimize Code
Output:

Future Scope

- Integration with Code Editors
- Real-Time Refactoring Suggestions
- Explainability Module
- Self-Hosted or Cloud API Offering

References

- **Vercel.** (2024). Next.js Documentation. Retrieved from
- https://nextjs.org/docs
- **Prettier Team.** (2024). Prettier: An Opinionated Code Formatter. Retrieved from
- https://prettier.io
- **ESLint.** (2024). Find and fix problems in your JavaScript code. Retrieved from
- https://eslint.org
- **Babel Contributors.** (2024). Babel: JavaScript Compiler. Retrieved from
- https://babeljs.io

Conclusion

- By blending AI with robust engineering practices, this tool not only simplifies the code refactoring process but also elevates development efficiency. Its privacy-first, scalable design makes it an indispensable asset for developers aiming to write cleaner, smarter, and more maintainable code.

- The modular architecture ensures scalability, while the stateless design prioritizes user privacy. By integrating services like language detection, compilation, analysis, and AI suggestions, the tool acts as a smart assistant for developers. It not only improves code readability and structure but also enhances overall productivity, making it a powerful companion in any modern development workflow.

Thank you !