World Journal of Advanced Engineering Technology and Sciences
eISSN: 2582-8266
Cross Ref DOI: 10.30574/wjaets
Journal homepage: https://wjaets.com/

(REVIEW ARTICLE)

Check for updates

Artificial Intelligence based code refactoring

Swathi Turai, Praneetha Potharaju, Rajasri Aishwarya Bepeta *, Mohammed Adil and Mani Charan Vangala

Department of CSE (Data Science), ACE Engineering College, Hyderabad, Telangana, India.

World Journal of Advanced Engineering Technology and Sciences, 2025, 15(02), 639-646

Publication history: Received on 26 March 2025; revised on 02 May 2025; accepted on 04 May 2025

Article DOI: https://doi.org/10.30574/wjaets.2025.15.2.0594

Abstract

One of the most difficult aspects of software development is maintaining and updating legacy code, which frequently requires a significant investment of time and energy to make the code more manageable, efficient, and readable. Using sophisticated AI, such as machine learning and large language models, the AI-Powered Codebase Refactorer is a clever tool made to make this process easier. It converts jumbled or out-of-date code—such as old Python or Java projects—into more organized, contemporary, and well-documented forms. The tool makes the code much easier to understand by adding useful comments and producing external API documentation in addition to applying best practices like modularization and design patterns to improve code structure. In order to make sure the code continues to function properly after changes, it uses automated tests and static analysis, which goes beyond simply tidying up syntax. Whether it's for system software, data tools, or web apps, this AI modifies its methodology to suit the particular project. Developers can reduce technical debt, save time, and maintain the functionality of critical software by automating a large portion of the refactoring process.

Keywords: Legacy Code Refactoring; AI-Powered Code Transformation; Large Language Models (LLMS); Static Code Analysis; Code Optimization; Machine Learning in Software Engineering; Code Maintainability

1. Introduction

Many businesses still rely on legacy systems, but maintaining them gets more difficult as time goes on. Outdated coding practices, deprecated software components, and inadequate documentation are frequently to blame for this. As a result, businesses deal with increasing maintenance expenses, delayed development schedules, and growing technical debt. Many of these systems were constructed with antiquated, monolithic designs that aren't appropriate for the scalable, fast-paced tech environments of today. Their overall complexity is increased by their inflexible architecture, which makes it challenging to incorporate new technologies or add contemporary features.

The AI-Powered Codebase Refactorer employs machine learning and artificial intelligence to address these problems and modernize legacy systems. Without affecting already-existing functionality, it automates the process of refactoring code, converting antiquated structures into versions that are cleaner, easier to maintain, and more scalable. By streamlining ineffective system components and optimizing code, the tool also improves performance. Its ability to produce thorough, understandable documentation that facilitates developers' work with the updated codebase is one of its main advantages. Businesses can modernize their systems more quickly and safely by adopting AI-driven automation, which will increase scalability, maintainability, and efficiency while lowering the amount of manual labor usually needed for such a complicated task.

* Corresponding author: Rajasri Aishwarya.

The AI-Powered Codebase Refactorer provides a novel way to address these problems. It automates the process of refactoring legacy code into a more scalable, modular, and maintainable form by utilizing machine learning and artificial intelligence. The tool improves the system's efficiency, performance, and structure while maintaining its current functionality. Additionally, it produces comprehensive and understandable documentation, which facilitates developers' comprehension and allows them to continue working with the updated code.

By automating time-consuming refactoring tasks, this project aims to improve maintainability by simplifying complex code, increase scalability through improved coding practices, and speed up the entire development process. AI-driven automation can help organizations modernize legacy systems with less risk and effort, facilitating a more seamless transition to modern architectures and lowering long-term operational burdens.

1.1. Problem Statement

Code produced by modern software development frequently lacks structure, is redundant, and deviates from best practices, which makes maintenance difficult and time-consuming. Deeper semantic and architectural problems are not addressed by traditional refactoring tools, which are mostly rule-based and have a narrow scope. In addition to being time-consuming, manual refactoring is prone to errors and requires developer expertise. To increase code quality, maintainability, and developer productivity, an intelligent, automated system that can comprehend code semantics and offer relevant, context-aware refactoring recommendations across several programming languages is desperately needed.

2. Literature review

Numerous studies have investigated the use of automated refactoring and artificial intelligence in the modernization of legacy systems. The two main areas are AI-assisted code refactoring, where machine learning models, especially deep learning and transformer-based approaches, analyze code syntax to recommend significant improvements, and static code analysis, where tools like SonarQube and ESLint identify technical debt and code smells to direct refactoring. Software maintainability has also been improved by the automatic generation of code documentation through the use of natural language processing (NLP) techniques. Through the detection and removal of system bottlenecks and the use of dependency analysis to convert monolithic architectures into scalable microservices, AI has also been crucial in performance optimization. Together, these developments enable AI-driven refactoring tools to automate code analysis, optimization, and restructuring. This reduces technical debt, improves modularity, and expedites the conversion of legacy systems to contemporary, secure, and efficient software infrastructures.

3. Existing methods

The majority of the code refactoring systems currently in use are manual or partially automated tools that are integrated into IDEs such as Visual Studio, Eclipse, or IntelliJ. These tools lack deeper analytical capabilities, but they work well for simple tasks like formatting code, extracting methods, and renaming variables. Their functionality is largely dependent on the developer's skill level; they provide little assistance when working with legacy systems or complex logic. Because of this, programmers must invest a great deal of time and mental energy, especially when working with large or old codebases where structural problems are difficult to spot.

Rule-based refactoring tools can carry out particular transformations, such as updating loop structures or enforcing coding standards, because they follow predefined instructions. But because they don't understand the code's context, their scope is naturally constrained. These tools are unable to identify project-specific subtleties, logical inconsistencies, or architectural inefficiencies. As a result, even though they help with superficial code enhancements, they frequently pass up chances for more significant and influential optimizations.

The static nature of traditional tools—their inability to learn or adapt through use—is another significant drawback. These tools provide the same general recommendations regardless of a project's size or complexity, disregarding past

trends or refactoring results. This results in uneven code quality, particularly in group settings. In the absence of a thoughtful, flexible strategy, these tools may unintentionally introduce errors or inconsistencies, compromising the codebase's overall dependability and maintainability.

4. Proposed method

The suggested system offers a code refactoring method powered by AI that greatly outperforms conventional tools. This system can analyse and comprehend code at a deep semantic level by utilizing sophisticated machine learning models like transformer architectures and graph neural networks. Instead of just making superficial adjustments, it finds intricate patterns, duplications, and inefficiencies and makes wise recommendations to enhance readability, performance, and maintainability. It can accurately refactor entire code blocks, rename variables, and reorganize logic while maintaining the program's original functionality.

This AI-based system's semantic understanding is one of its main advantages. The system can learn more about the structural and functional aspects of a program by examining its Abstract Syntax Tree (AST). This enables it to identify architectural problems that simple line-by-line tools frequently overlook, like tightly coupled modules or bloated functions. The system learns what makes code good and provides context-aware refactoring recommendations that go beyond static rule sets after being trained on large datasets from real-world codebases.

Its ability to integrate seamlessly and support multiple languages is another significant benefit. The system is made to comprehend language-specific syntax and best practices, regardless of whether the developer is working with Python, Java, C++, or JavaScript. REST APIs and IDE plugins make integration simple, enabling developers to seamlessly integrate the tool into their current workflows. Furthermore, by keeping developers in charge, features like rollback options, side-by-side code comparisons, and refactoring previews promote transparency and trust in the refactoring process.

The design of this system also prioritizes scalability and user experience. It is appropriate for enterprise-level applications since it is designed to manage sizable codebases and multiple users at once. The output is assessed and optimized using performance metrics like the Maintainability Index and Cyclomatic Complexity, guaranteeing significant gains. The system makes code modernization more efficient, dependable, and scalable without requiring knowledge of artificial intelligence or machine learning thanks to its user-friendly interface and one-click automation.

5. Methodology

5.1. Data Collection

Training data should come from clean, high-quality repositories on GitHub and CodeSearchNet, focusing on Python and Java. These should follow established coding standards to ensure reliable and maintainable code.

5.2. Preprocessing

Tree-sitter is used to parse code into Abstract Syntax Trees (ASTs), and models such as CodeBERT or CodeT5 are used to tokenize the code for embedding. Code that smells like lengthy methods or duplicates is marked for improvement.

5.3. Model Training

Semantics and structure of code are analyzed by machine learning models like Graph Neural Networks (GNNs) and CodeBERT. To learn efficient code improvements, the model is refined using labeled refactoring examples.

5.4. Integration

For effective refactoring, the system integrates rule-based engines and machine learning. While machine learning models address complex code smells, rules handle simple tasks.

5.5. Architecture

An IDE plugin or a Next.js web interface can serve as the frontend. The backend communicates via RESTful APIs and makes use of Flask or FastAPI with modules such as Language Detector and RefactorEngine.

5.6. Deployment

The MVP is a Python refactoring plugin for Visual Studio Code. Version 2 will support cloud infrastructure such as AWS or GCP for scalability, as well as Java and CI/CD pipelines.

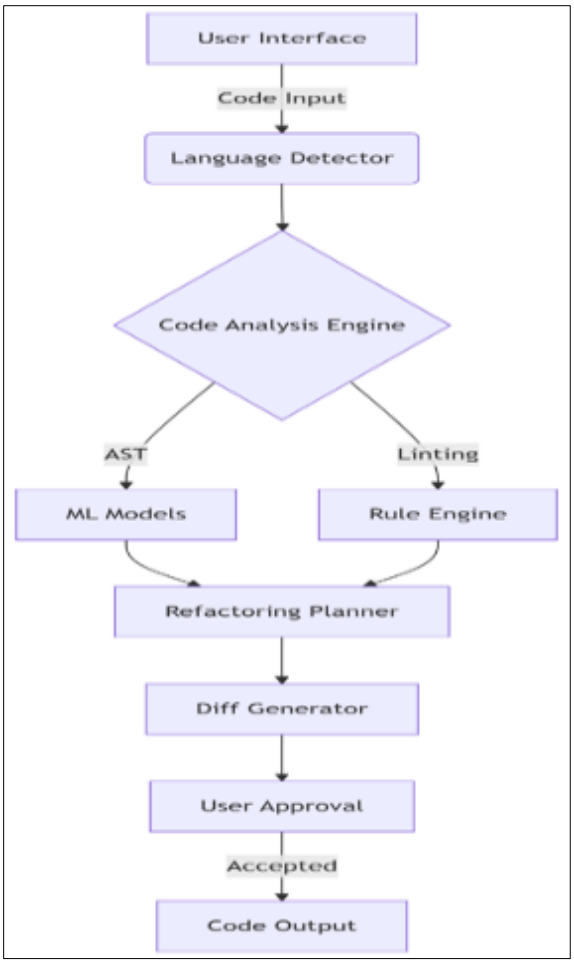

Figure 1 Methodology

5.7. System Architecture

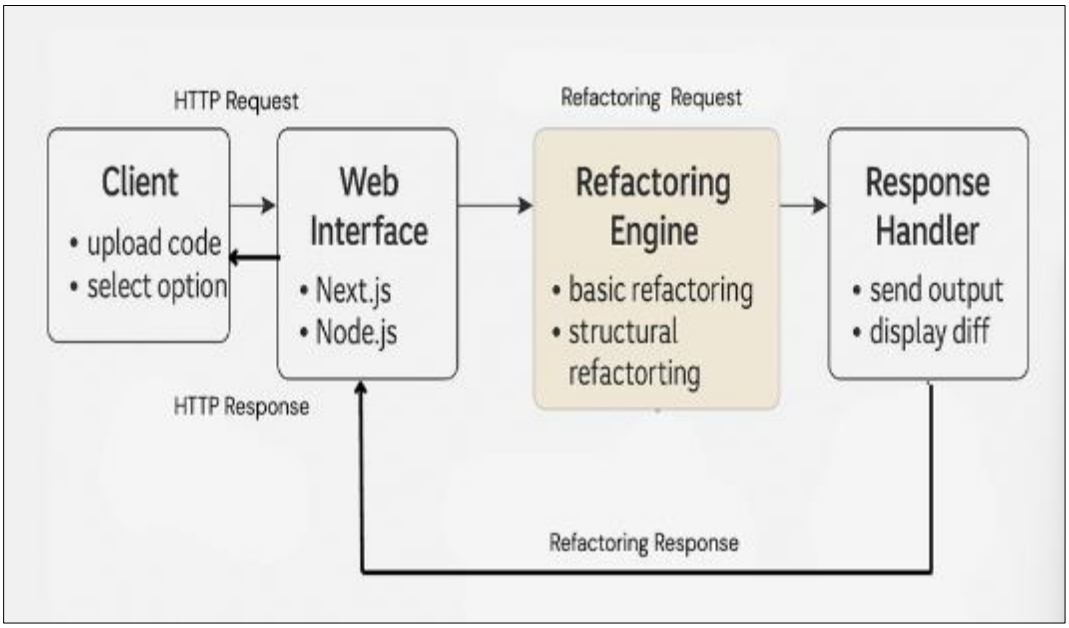

Figure 2 System Architectur

5.8. System Components

5.8.1. Frontend Module

React and TypeScript are combined in Next.js to create an intuitive user interface for the frontend. Users can choose the preferred programming language, enter source code, and see the optimized result. The user can view the refactored code in a readable format after the backend returns it.

5.8.2. Backend API Module

The backend, which was created in Python using FastAPI, manages communication between the machine learning model and the frontend. After validating the code and language inputs, it sends the information to the machine learning model. The response is prepared by the backend and sent back to the frontend after the optimized code has been generated.

5.8.3. Machine Learning Model Module

The main refactoring is done by this module. It makes use of pretrained models, such as Code Llama or StarCoder, that have been refined using examples of badly written and well-refactored code. The model, which was created using Hugging Face and PyTorch, produces code that is faster, cleaner, and uses less memory without changing the original logic.

5.8.4. Evaluation Module

The system uses a number of automated metrics to assess the output following refactoring. These include cyclomatic complexity to gauge code simplicity, runtime and memory usage improvements, and BLEU and ROUGE scores for code similarity. This guarantees that the code produced is effective and of excellent quality.

5.8.5. Post-Processing Module

The output code is run through code formatters such as Black (for Python) or Prettier (for JavaScript) and linters are applied to guarantee consistency and readability. In this step, the syntax is cleaned up, minor bugs are fixed, and the user is given polished, standardized code.

5.8.6. Storage Module

The optimized output, associated metadata, and the original input are all stored in a MongoDB database. This enables the system to keep track of prior refactoring operations, and the information gathered can also be utilized to enhance the model in subsequent iterations.

5.8.7. System Workflow

The user submits code via the frontend to start the process. After processing and validating the input, the backend forwards it to the machine learning model and sets up the environment as required. The code is restructured by the ML model, which enhances both its structure and performance. The evaluation module confirms quality, and post-processing guarantees clean output. The polished code is then sent back to the frontend by the backend for display.

6. Results and Discussion

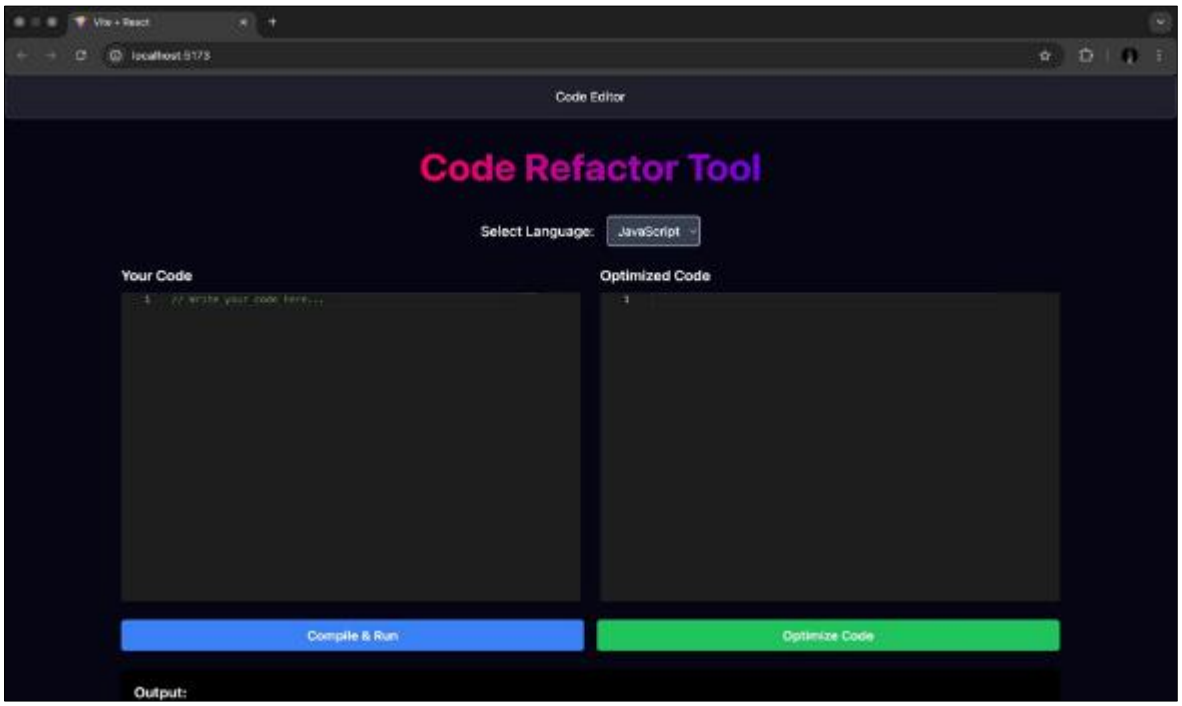

Figure 3 Interface

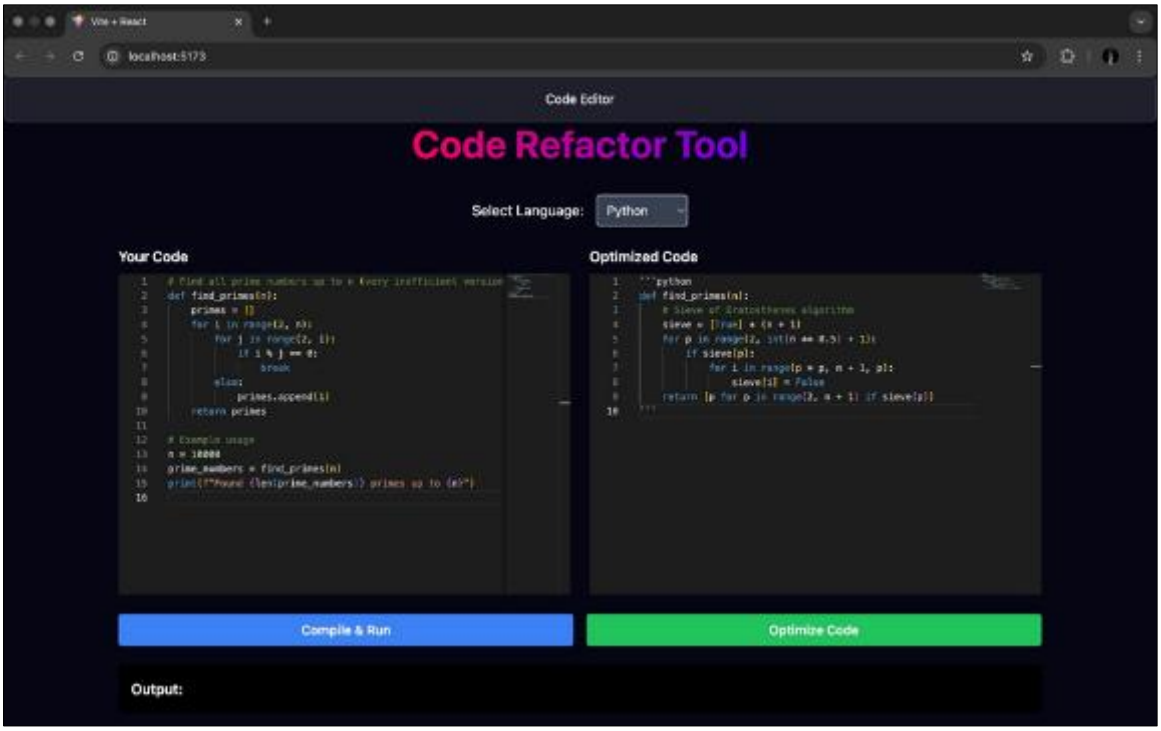

Figure 4 Output

7. Conclusion

Through the use of artificial intelligence, machine learning, and natural language processing, the AI-Based Code Refactoring project presents a novel, intelligent system that improves and automates the software refactoring process. In contrast to conventional rule-based tools, this system makes use of deep learning models, like Transformers and Graph Neural Networks, to comprehend the semantic structure of code. This allows it to identify intricate problems, such as anti-patterns and architectural flaws. It guarantees precise and secure code transformations by supporting a variety of programming languages (such as Python, Java, and JavaScript) via abstract syntax trees (ASTs). With an interactive interface, IDE integration, and Git support, the system also emphasizes a developer-centric workflow, enabling developers to preview and manage changes while preserving traceability.

The system, which is built for both scalability and performance, can scale through cloud deployment to meet enterprise needs while processing small to medium codebases with efficiency. To guarantee quality and dependability, refactored code is assessed using common metrics such as the Cyclomatic Complexity and Maintainability Index and validated using automated test cases. Future improvements could include team-based collaboration tools, explainable AI to support recommendations, real-time refactoring in live coding environments, and customizable AI models. All things considered, this project demonstrates how AI can enable programmers to enhance code quality, lower technical debt, and create software systems that are easier to maintain.

Compliance with ethical standards

Disclosure of conflict of interest

There is no conflict of interest.

References

[1] Vercel. (2024) Next.js Documentation Comprehensive guide for building React applications with Next.js. https://nextjs.org/docs

[2] Prettier Team. (2024) An opinionated code formatter supporting multiple languages. https://prettier.io

[3] ESLint. (2024) A static code analysis tool for identifying problematic patterns in JavaScript code. https://eslint.org

[4] Babel Contributors. (2024) A JavaScript compiler that allows you to use next-generation JavaScript. https://babeljs.io

[5] Refactoring: Improving the Design of Existing Code By Martin Fowler (1999) – A seminal book on code refactoring techniques. https://martinfowler.com/books/refactoring.html

[6] Code Complete: A Practical Handbook of Software Construction By Steve McConnell (2004) – Comprehensive guide on software construction best practices. https://en.wikipedia.org/wiki/Code_Complete

[7] Mining Source Code Repositories at Scale: From Control Flow Graphs to Deep Learning By Miltiadis Allamanis & Charles Sutton (2013) – Discusses large-scale mining of source code using deep learning. https://homepages.inf.ed.ac.uk/csutton/publications/MiningSourceCodeAtScale.pdf

[8] Deep Learning for Code Generation By Guillaume Lample & François Charton (2020) – Explores deep learning techniques for generating code. https://arxiv.org/abs/2005.13981

[9] AI for Code: The Future of Software Development By Brian Smith (2020) – Explores the impact of AI on software development practices. https://www.scirp.org/journal/paperinformation.aspx?paperid=10117

[10] Code Reuse in Open-Source Software By Stefan Haefliger , Georg von Krogh, & Sebastian Spaeth (2008) – Discusses patterns in open-source codebases that support AI training for refactoring tools. https://www.researchgate.net/publication/220422146_Code_Reuse_in_Open_Source_Software

Author's short biography

Mrs. Swathi Turai Mrs. Swathi Turai, Department of CSE (Data Science), Ace Engineering College, Affiliated to JNTUH Ghatkesar, Hyderabad, India. She has been guided for Mini and Major projects for different pass out batches. The research papers are published with respect to them also. Participated and Attended various Workshop, Faculty Development Programs conducted at intra level and Inter level enhanced the knowledge in Machine Learning, Deep Learning, Emerging Technologies, DBMS, Web Technologies. Her Research Areas Includes problem solving through C and Python programming, Web Technologies, Machine Learning, Artificial Intelligence. Received a certificate of appreciation from NPTEL.	
Praneetha Potharaju A final-year B.Tech student at ACE Engineering College, specializing in Computer Science and Engineering (Data Science). I am passionate about data science and programming; I enjoy discovering emerging technologies and expanding my expertise. I am committed to continuously improving my skills and leveraging them to solve real-world challenges in my field.	
Rajasri Aishwarya Bepeta A final-year B.Tech student at ACE Engineering College, specializing in Computer Science and Engineering (Data Science). I have a keen interest in data science and programming, constantly exploring new technologies to enhance my knowledge and skills. My goal is to apply my expertise effectively in real-world scenarios.	
Mohammed Adil A final-year B.Tech student at ACE Engineering College, specializing in Computer Science and Engineering (Data Science). I love diving into data science, programming, and emerging technologies. Exploring new concepts and refining my skills excites me, and I'm eager to apply my knowledge to solve meaningful challenges.	
Mani Charan Vangala A final-year B.Tech student at ACE Engineering College, specializing in Computer Science and Engineering (Data Science). I am passionate about programming and data-driven solutions. I enjoy learning about innovative technologies and continuously developing my skills to make a meaningful impact in the field.