**Capstone Project Proposal Report**

**(Individual Report)**

**Instructions:**

|  |  |  |
| --- | --- | --- |
| Guide Approval (initials/date): | Dr. Hari Kishan Kondaveeti | August 27, 2025 |

**CAP4001– Capstone Project Proposal Report**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Student Name** | | **Sahil Sharma** | | |
| **Student Register Number** | | **22BCE7921** | | |
| **Programme** | | B. Tech CSE Core | | |
| **Semester/Year** | | Fall 2025-26 | | |
| **Guide(s)** | | Dr. Hari Kishan Kondaveeti | | |
| **Project Title** | | **Automated Codebase Refactoring** | | |
|  | | | | |
| **Reg. No** | **Name** | | **Major** | **Specialization** |
| 22BCE7921 | Sahil Sharma | | Computer Science | Core |
| 22BCE7591 | Harsh Raj | | Computer Science | Core |
|  |  | |  |  |
|  |  | |  |  |

### **Project and Task Description**

#### **(a) Project Summary**

The **Codebase Refactoring Project** is an AI-powered system designed to analyze, refactor, and document messy codebases while generating interview-style questions. This project addresses a common challenge faced by developers, educators, and recruiters: navigating poorly documented code repositories and evaluating technical understanding.

**Codebase:** A codebase is a collection of source codes for a software project, that may include program files, configurations, documentation, and tests. It serves as the source of truth that developers collaborate on, usually managed with version control systems like Git.

**Problems with unorganized/raw codebases developed by students/programmers:**

* **Poor Documentation** – Developers struggle to understand or extend the code.
* **High Complexity** – Messy, unstructured code increases bugs and slows progress.
* **Low Readability** – Inconsistent style, unclear variable names, and lack of comments reduce clarity.
* **Technical Debt** – Quick fixes and outdated practices lower long-term quality.
* **Scalability Issues** – Large codebases become difficult to test, refactor, and extend.
* **Security Vulnerabilities** – Weak coding practices or outdated dependencies expose the system to risks.
* **Knowledge Transfer & Assessment Challenges** – New developers or students find it hard to revise and memorize logic and concepts, making evaluation and learning difficult, especially for interviews.

**The problem our project aims to solve:**

* Automate refactoring of messy, unstructured code to reduce complexity.
* Improve readability and documentation with inline comments and summaries.
* Support learning and assessment by generating interview-style technical questions.

**Purpose**

The purpose of this project is to save time and effort by using Artificial Intelligence to understand large codebases and improve them. Instead of manually reading every file, our project will:

1. Analyze the project’s structure.
2. Clean and reorganize the code.
3. Write simple explanations and documentation.
4. Generate useful interview-style questions about the project.

This tool benefits:

* **Students:** Learn better by seeing clean, documented code and practice with code-related questions.
* **Developers:** Quickly understand large repositories and maintain them.
* **Recruiters/Educators:** Automatically create technical questions for interviews or exams.

**Specifications:**

* **Input Options:** Users upload a **zip file** of their code, paste raw code, or provide a **GitHub repository link** (an online platform where developers store and share code).
* **Output:** Refactored code (A cleaned-up version of the code with better structure) with inline comments, documentation, and 5 tailored interview questions.

#### **Core Features (Explained)**

1. **Code Understanding & Refactoring**:
   * The system reads and analyzes the project structure (folders, files, and functions).
   * It reorganizes messy code without changing its functionality.
   * Adds helpful comments and improves formatting.
2. **Question Generation**:
   * AI creates interview-style questions based on the code logic, helping students and interviewees.
3. **Automated Documentation**:
   * Generates human-readable summaries for files and functions.

4. **Tech Stack:**

* + **Frontend:** React/Next.js web interface.
  + **Backend:** Python with FastAPIs.
  + **AI Core:** Large Language Models (LLMs) like Gemini Pro, Grok API, or OpenRouter – AI models used to understand and rewrite code. Model selection will be done by analysing their performance matrix.
  + **Integrations:** GitHub API and vector databases for embeddings.
  + **Deployment:** Vercel (for frontend) and Render (for backend).

**Summary of Approach:**

The system follows a three-phase workflow:

1. **Input Phase:** User uploads a repository or topic.
2. **Processing Phase:** File Tree Walker → Code Analyzer → Refactorer → Question Generator.
3. **Output Phase:** Provide refactored, documented code and an interview preparation module.

### 

### **(b) Individual Role and Tasks:**

**1)** **Sahil Sharma – 22BCE7921 (Backend Engineer & AI Architect)**

* Builds backend services that analyse and refactor code.
* Integrates AI models for code understanding and question generation.
* Designs APIs (ways for frontend and backend to communicate)

**2)** **Harsh Raj – 22BCE7591 (Frontend Developer & Deployment Lead)**

* Creates evaluation metrics to test system accuracy.
* Designs and builds a simple, clean user interface with Next.js.
* Ensures frontend and backend integration.
* Deploys the app on Vercel and Render for public access.
* Writes documentation for frontend workflows.

**Deliverables:**

1. A functional and responsive frontend.
2. Smooth integration between frontend and backend APIs.
3. Deployment setup on Vercel (frontend) and Render (backend).
4. Performance metrics, test cases and debugging reports to ensure system stability.
5. Clear documentation of frontend workflows and deployment processes.

#### **(c) Detailed Approach**

1. **Frontend Development:**
   * Build a clean, user-friendly interface with Next.js/React.
   * Ensure proper integration with backend APIs for seamless user experience.
2. **Deployment (Vercel & Render):**
   * Deploy frontend on Vercel with global edge delivery for performance.
   * Deploy backend services on Render and configure secure communication.
3. **Testing & Debugging:**
   * Create automated test cases for frontend modules.
   * Debug integration issues between frontend and backend.
   * Evaluate different combination of approaches and models based on developed performance measures.
4. **Documentation:**
   * Write user-friendly guides for deployment steps and frontend workflows.
   * Maintain technical notes for future scalability.

#### **(d) Phases of Design Process**

|  |  |  |
| --- | --- | --- |
| Phase/Review | Objective | Deadline |
| Review 1 | To submit a detailed project proposal that explains the problem statement and proposed solution; present a clear approach for implementation with defined steps; outline measurable criteria for evaluating the performance of the Large Language Model (LLM); and create a PowerPoint presentation summarizing the overall project overview.. | 30th August, 2025 |
| Review 2 | To deliver a functional project prototype that integrates a working Large Language Model (LLM) with core backend features, demonstrates key functionalities such as code analysis and question generation, and includes a basic but usable user interface (UI) for testing and feedback. | 30th September 2025 |
| Review 3 | To present a fully deployed version of the project with a refined, visually appealing, and user-friendly interface, ensuring smooth integration between the frontend, backend, and LLM-based functionalities. | 8th November 2025 |
| Review 4 | To submit the final project report along with all required supporting materials, including a project poster, deployment files, and source code ensuring the project is well-documented and presentation-ready. | 5th December 2025 |

**Outcome Matrix:**

|  |  |
| --- | --- |
| **Outcomes:** | **Plan for demonstrating outcome:** |
| a) an ability to apply knowledge of mathematics, science, and engineering | This project uses **computer science fundamentals** to solve a real problem: for understanding large, messy codebases. We use:  • **Static code analysis** (studying code without running it) to see how the program is structured.  • **Tree traversal** (a method of visiting nodes in a tree-like structure) to read file and folder hierarchies of a project.  • **Graph-based repository mapping** (representing files and their relationships like a graph of nodes and connections).  • **Software engineering principles** like modularity (breaking the project into smaller, reusable parts) and maintainability (making it easy to update in the future).  • **AI models** to understand code logic, rewrite it neatly, and generate related technical questions. |
| c) an ability to design a system, component, or process to meet desired needs within realistic constraints such as economic, environmental, social, political, ethical, health and safety, manufacturability, and sustainability | The system is built in a way that is practical, affordable, and future-ready. We:  • Use **open-source tools** (free software libraries) to avoid high costs.  • Keep the backend (server-side system) **lightweight** (a server-side system built with minimal code, fewer dependencies, and efficient frameworks so it runs fast, uses less memory) so it does not need expensive servers.  • Design it in a **modular** way (easy to change one part without breaking the whole).  • Use **containerization** (packaging the application with its dependencies using Docker) so it can easily run on different machines or cloud platforms. |
| d) an ability to function on multidisciplinary teams | We work as a small team with clear roles: I focus on backend programming and AI integration, while my teammate works on frontend (user interface) and deployment. We use **GitHub** (a version control platform) to share code, track changes, and collaborate. We follow **Agile methodology** (working in small, iterative steps called sprints instead of building everything at once). |
| e) an ability to identify, formulate, and solve engineering problems | The project solves a **real-world problem**: developers struggle with understanding large, messy, undocumented projects. We solve this by:  • Building a tool to read and map large repositories.  • Supporting multiple programming languages.  • Automatically generating useful questions to help students and interview candidates. |
| g) an ability to communicate effectively | We ensure clear communication through:  • **Readable documentation** (step-by-step guides and explanations).  • **Inline code comments** explaining what each function does.  • **Clear APIs** (ways for the frontend to interact with the backend).  • Sprint meetings to update progress and ensure teamwork. |
| k) an ability to use the techniques, skills, and modern engineering tools necessary for engineering practice | This system uses:  • **FastAPI** (a Python framework for building APIs).  • **Docker** (to run the system in a portable container).  • **GitHub API** (to interact with repositories programmatically).  • **Large Language Models (LLMs)** (AI models that understand and rewrite code).  Future improvements include adding **CI/CD pipelines** (Continuous Integration/Continuous Deployment—automating testing and software updates) for a smoother development process. |

**Realistic Constraints:**

Thisproject considers multiple practical constraints to ensure the system’s feasibility and reliability:

1. **Economic Constraints:**  
   We use **free, open-source software** to keep costs low. Cloud hosting is optional, and the system’s lightweight design avoids expensive hardware.
2. **Environmental Constraints:**  
   We reduce wasteful computing by **caching** (saving previous results) and avoiding unnecessary re-processing. This cuts down on energy use.
3. **Social Constraints:**  
   The tool helps **students, developers, and recruiters** by making code easier to understand and prepare for interviews. This can close skill gaps and make hiring more efficient.
4. **Ethical Constraints:**  
   Uploaded code is **not stored or shared** without permission, keeping user data safe. We avoid plagiarism by only analyzing the code for educational purposes.
5. **Health and Safety Constraints:**  
   Since this is software, there are no physical safety risks. The interface is kept **simple and stress-free** for the users.
6. **Manufacturability & Sustainability:**  
   Because the system is **modular**, we can add new programming language support in the future. Using open-source tools ensures long-term availability and community support.

**Engineering Standards:**

The project follows established software engineering and AI standards to ensure quality, maintainability, and interoperability:

1. **Software Engineering Standards:**  
   We follow **modular design** (building in parts), **Agile methodology** (small, iterative improvements), and use **GitHub version control** to manage teamwork.
2. **Coding & Documentation Standards:**  
   We write Python code following **PEP 8 style guidelines** (a standard for writing neat Python code). Code has **inline comments** and we write **API documentation** (instructions on how to use system features).
3. **AI & Data Handling Standards:**  
   We use AI responsibly: no data is kept without permission, and all outputs are transparent.
4. **Security Practices:**  
   Basic safety measures: restricting data access, using safe API calls, and isolating user data during analysis to prevent leaks.

**Performance Measures:**

**Evaluation Framework for Comparing LLM APIs:**

The team will evaluate different APIs such as Grok, Claude, Gemini, and GPT on a set of carefully designed prompts. To make this evaluation systematic, the team will build a matrix of prompts × models, and rated each output based on multiple performance measures.

**Matrix Setup:**

* Rows: Different prompts/tasks relevant to our project, such as *code refactoring, documentation generation, code comprehension, and interview question creation*.
* Columns: Different models (Grok, Claude, Gemini, GPT, etc. and their versions/releases).
* Cells: The objective and subjective scores of each model for the respective prompt.

This structure allows the team to make direct, side-by-side comparisons of performance.

Since the project focuses on automated codebase refactoring and comprehension, we have defined performance measures in two categories:

1. **Automatic Metrics:**

* **BLEU / ROUGE** – For comparing generated documentation against a set reference text and generating a score.
  + BLEU (Bilingual Evaluation Understudy) measures how much the machine-generated text overlaps with a reference text at the *n-gram* (word sequence) level.
  + ROUGE (Recall-Oriented Understudy for Gisting Evaluation) → measures how much of the reference text’s content is captured in the model’s output.
* **Code Quality Linters** – To measure how clean or standardized the refactored code is (using tools like Pylint, ESLint, etc.). They evaluate style, formatting, and adherence to best practices and score.
* **Test Case Accuracy** – To verify correctness of outputs on provided test cases. Validates functional correctness of the refactored code. Scores on how well logic is preserved.

**b) Human Evaluation Rubric (Subjective Quality):** Human volunteers independently rate outputs on a **Likert scale (1–5)** for:

* Correctness
* Readability
* Relevance of the generated comment
* Relevance of the generated interview questions
* Conciseness

An average of this score will be considered for evaluation a particular matrix combination.