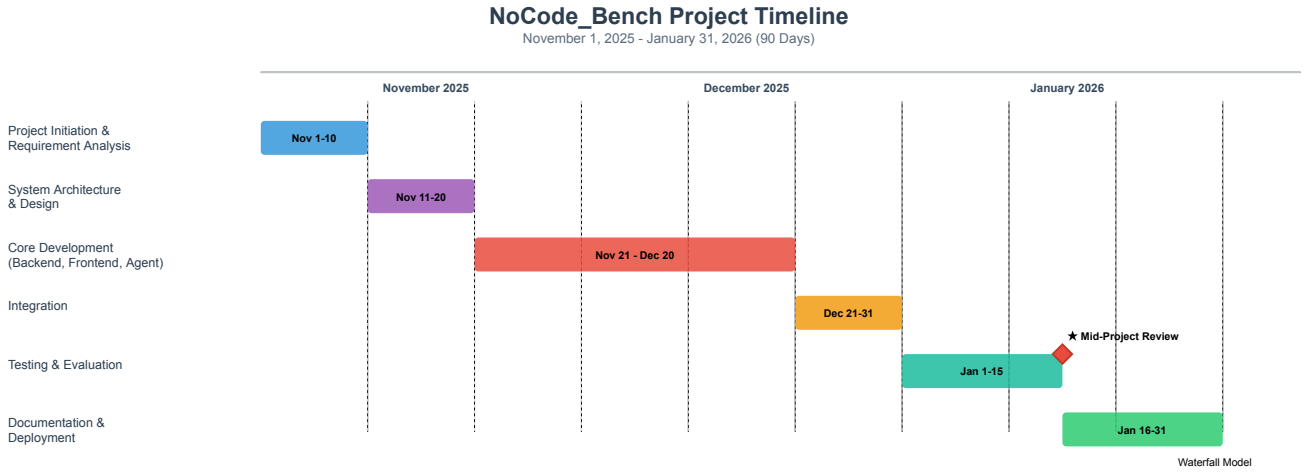
Project Management Report

Table of Contents

1. Project Timeline & Milestones

Project: NoCode_Bench
Start Date: November 1, 2025
End Date: January 31, 2026
Total Duration: 90 Days
Key Milestone: January 15, 2026 (Mid-Project Review)

Date	Tasks
2025.11.01 - 2025.11.10	Project Initiation & Requirement Analysis
2025.11.11 - 2025.11.20	System Architecture & Design
2025.11.21 - 2025.12.20	Core Development - Backend & Frontend & Agent
2025.12.21 - 2025.12.31	Integration
2026.01.01 - 2026.01.15	Testing & Evaluation
2026.01.16 - 2026.01.31	Documentation & Deployment



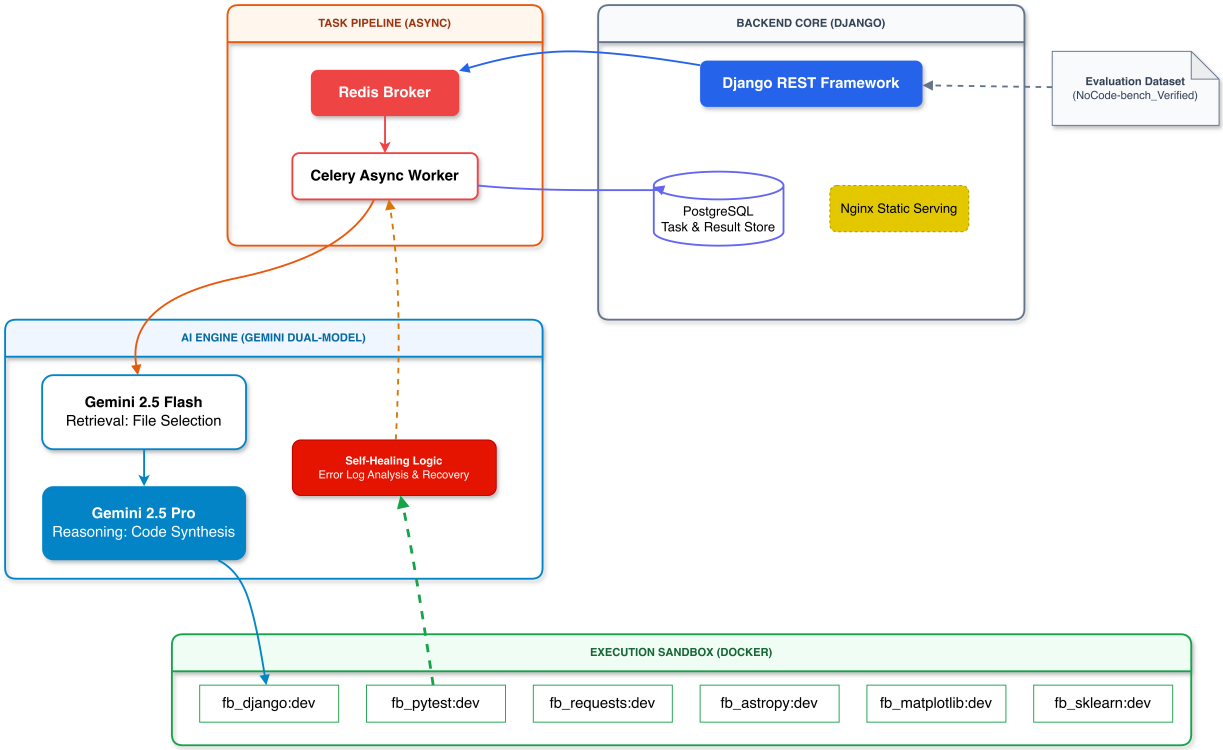
2. System Architecture

The NoCode-bench system is built on a distributed, asynchronous architecture designed to handle documentation-driven code generation and containerized validation. The architecture consists of four primary layers:

- **Web & API Layer**: A Django-based backend provides RESTful endpoints for task specification, allowing users to submit either verified benchmark tasks or custom GitHub repository URLs.
- **Task Orchestration Layer**: Utilizes Celery with a Redis message broker to manage long-running evaluation tasks asynchronously. This ensures the web interface remains responsive while the agent processes complex code changes.
- **AI Agent Core**: This component integrates the Gemini-2.5-Flash model for efficient file retrieval and the Gemini-2.5-Pro model for sophisticated code generation. It manages a

self-correction loop by maintaining a history of failed attempts and error logs to refine subsequent code patches.

- **Isolated Execution Layer**: A Docker-based environment where generated patches are applied to cloned repositories. It executes pre-configured test suites (Fail-to-Pass and Pass-to-Pass) within specialized containers to verify the functional correctness of the AI-generated code.



The NoCode-bench pipeline automates the transition from a natural-language documentation change to a verified code implementation through a structured multi-stage process:

1. **Environment Initialization**:
    - The system creates a isolated workspace by cloning the target repository or copying a verified codebase into a temporary directory.
    - Git is initialized within the workspace to track changes and generate patches.
2. **AI-Driven File Retrieval**:
    - The system scans the repository for relevant file paths (e.g., `.py`, `.html`, `.js`).
    - **Gemini-2.5-Flash** identifies core files necessary for the change, and the system extracts their content up to a 200,000-character limit to form the LLM context.
3. **Patch Generation**:
    - **Gemini-2.5-Pro** processes the documentation change and file contents to generate new code.
    - The agent is instructed to provide full file contents using a specific delimiter format to ensure successful application to the workspace.
4. **Containerized Validation**:

- The generated patch is transferred to a pre-configured Docker container matching the project's environment (e.g., Django, scikit-learn).
- The system applies the code patch alongside a specific "feature test patch" and executes the project's test suite using `conda` or `pytest`.

5. **Iterative Self-Correction**:
    - If the tests fail, the system captures the error logs and includes them in the history for a subsequent attempt.
    - The agent analyzes these errors to refine the logic and generate an updated patch.
6. **Results & Metrics Finalization**:
    - The system calculates performance metrics, including Fail-to-Pass (F2P) and Pass-to-Pass (P2P) success rates, token usage, and execution time.
    - The final status (e.g., `COMPLETED`, `FAILED_TEST`) and the generated patch are saved to the database for user inspection.

3. Method

Component	Choice	Rationale
File Retrieval	LLM-based Semantic Filtering (Gemini-2.5-Flash) with Path Scanning	Uses high-speed LLM reasoning to identify core files from a large codebase (up to 3000 paths) before fetching full content, ensuring the context remains relevant and within token limits.
Code Generation	Full-file Rewrite Strategy via Gemini-2.5-Pro	Avoids the complexities of parsing diffs by generating entire file contents with structured delimiters, ensuring code consistency and easier application to the workspace.
Docker Validation	Isolated Containerized Testing (F2P & P2P)	Provides a reproducible and safe environment to verify patches by running existing tests (Pass-to-Pass) and new feature tests (Fail-to-Pass) to ensure functional correctness without regressions.
Self-Correction Loop	Iterative Error-Feedback Loop (Attempt-based)	Captures execution failures and test logs to feed back into the model's context, allowing the agent to analyze logic errors and refine its implementation in subsequent attempts.

4. Team Roles & Responsibilities

Name	Role	Key Contributions

Name	Role	Key Contributions
Kai-Hao, Yang	Project Lead & AI Architect	Led the overall project direction and system design; architected and orchestrated agentic workflows across the platform.
Hao Lin	Backend & Evaluation Engineer	Test and engineered the automated evaluation pipeline.
Han Hu	Infrastructure & DevOps Engineer	Built Docker sandboxes and automated benchmark environment deployment.
Kaihui, You	Frontend & UI/UX Designer	Design and implement UI components and page layouts, Handle styling and responsive design.
Hsuan Lien	Frontend & API Integration Engineer	Integrate frontend with backend APIs, Handle routing, and error handling.

5. Future Plans

To enhance the performance and versatility of the NoCode-bench platform, several key technical improvements are planned based on the current system's architecture:

- **Dynamic Multi-Attempt Self-Correction**: The current system is limited to a single generation attempt per task (`MAX_ATTEMPTS = 1`). Future versions will implement a dynamic budget for self-correction, allowing the agent to perform multiple iterative fixes based on more granular feedback from test failure logs.
- **Enhanced Retrieval for Large-Scale Codebases**: Currently, the file retrieval component scans a maximum of 3,000 files and is restricted by a 200,000-character context limit. We plan to implement a more scalable RAG (Retrieval-Augmented Generation) approach or a hierarchical directory-traversal agent to handle enterprise-level repositories.
- **Broadening Framework and Language Support**: The system is currently optimized for Python-based repositories like Django and scikit-learn. Future updates will include specialized Docker environments and configuration mappings for other ecosystems such as JavaScript (Node.js), Go, and Java to evaluate cross-language documentation-driven coding.
- **Advanced Multi-Agent Collaboration**: The architecture currently uses a two-stage sequential pipeline (Flash for search, Pro for coding). We aim to transition to a multi-agent framework where specialized agents (e.g., a 'Reviewer Agent' and a 'Debugger Agent') collaborate to verify code quality and security before the final execution in Docker.
- **Integration of Real-Time Evaluation Metrics**: Beyond the existing Fail-to-Pass (F2P) and Pass-to-Pass (P2P) metrics, we plan to incorporate code complexity analysis and token efficiency tracking directly into the results dashboard to provide deeper insights into the agent's implementation efficiency.